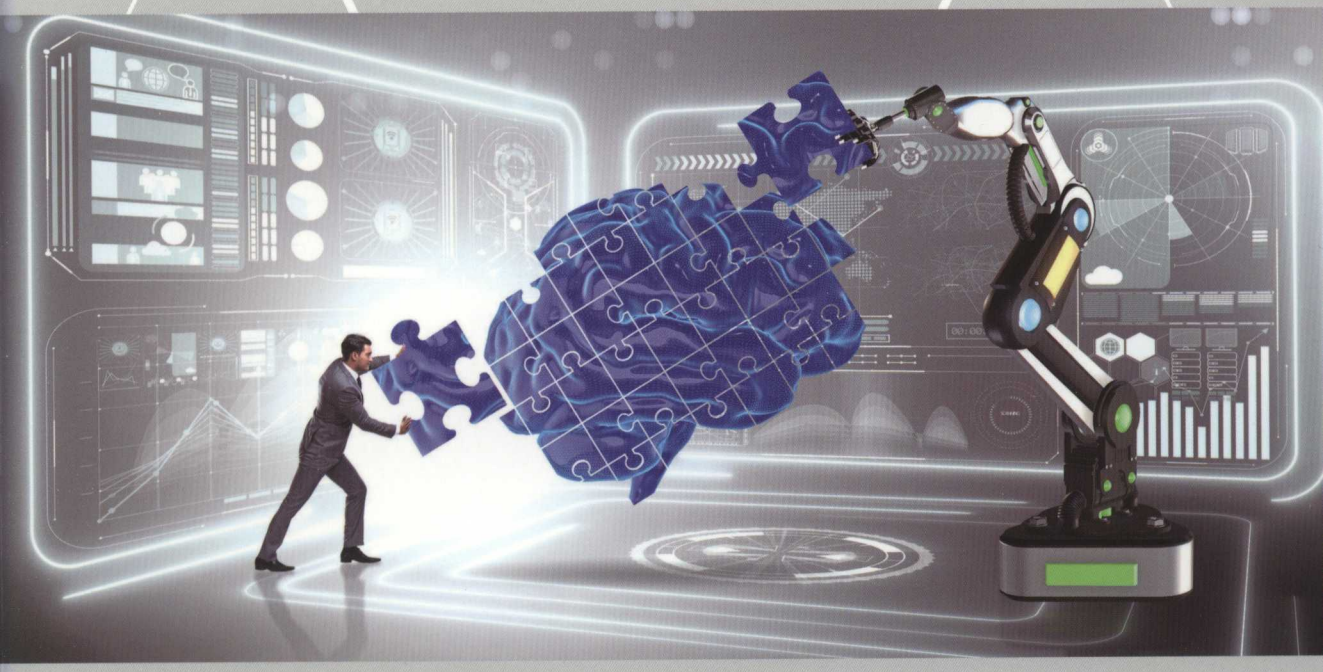
异步图书
www.epubit.com.cn

自己动手写神经网络

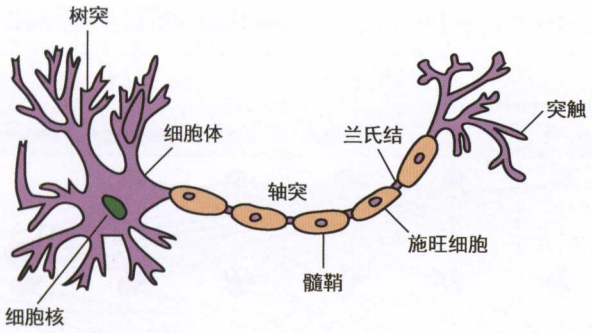
葛一鸣◎著

ZIJI DONGSHOU XIE
SHENJING WANGLUO

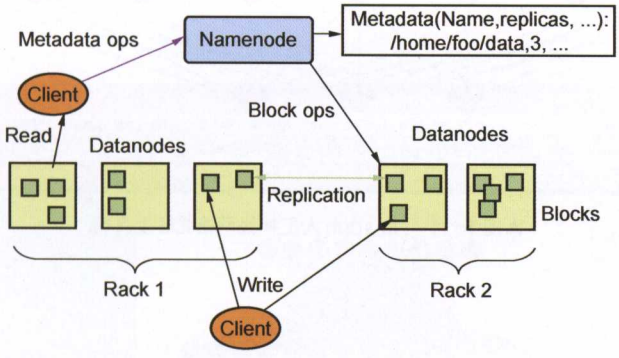


AlphaGo 战胜李世石，标志着新一轮人工智能的浪潮已经来袭。
而你是否已经做好迎接新的人工智能技术的准备？
本书，将带你一探作为 AlphaGo 基石的人工神经网络。
本书，不局限于纸上谈兵，我们用代码诠释一切。
本书，可能改变你对人工智能的态度。

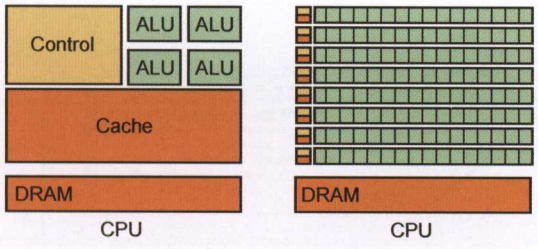
中国工信出版集团
人民邮电出版社
POSTS & TELECOM PRESS



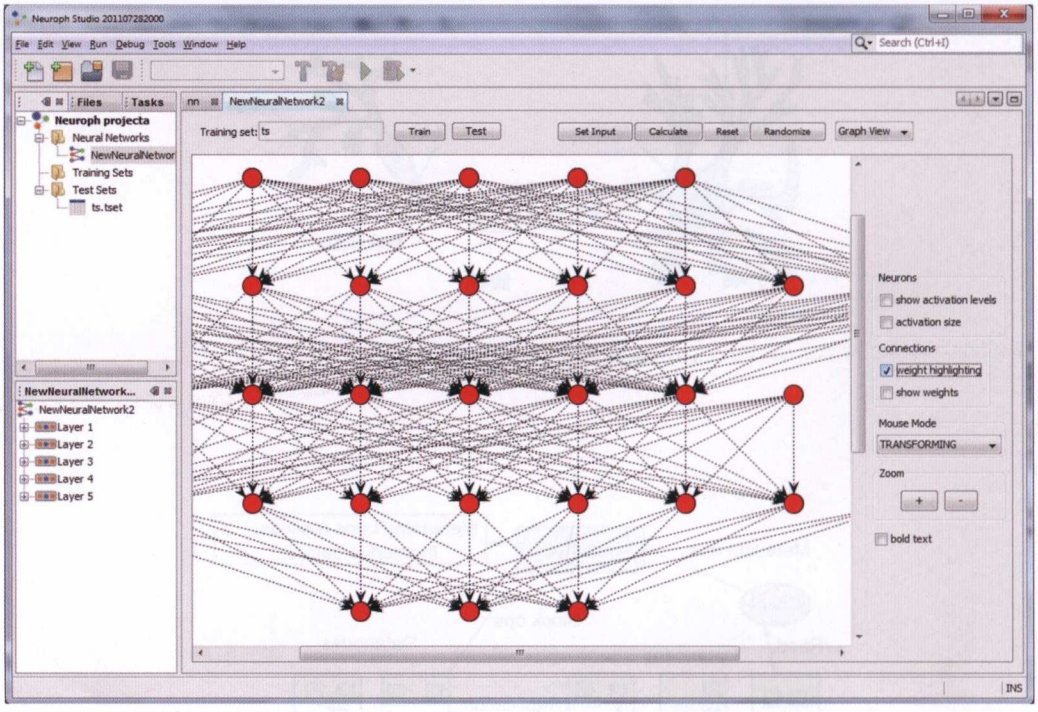
▲图1-8　神经细胞的结构



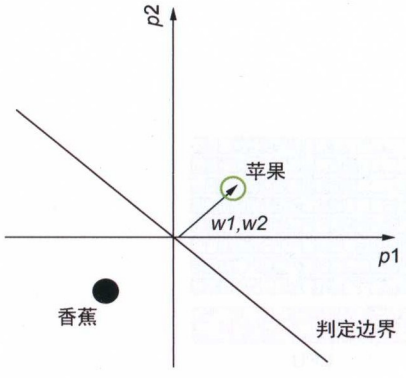
▲图1-9　Hadoop分布式文件系统的架构



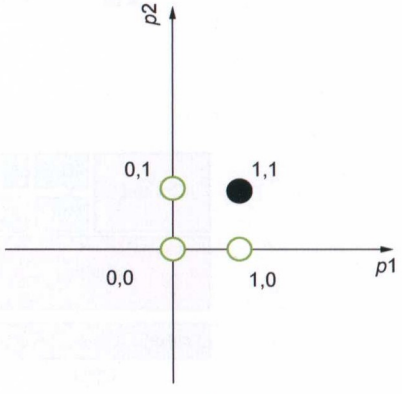
▲图1-10　CPU和GPU的架构差别



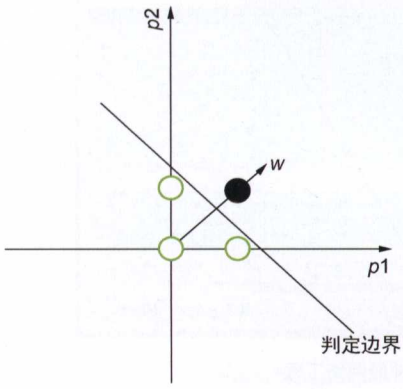
▲图1-11　Neuroph人工神经网络IDE工具箱



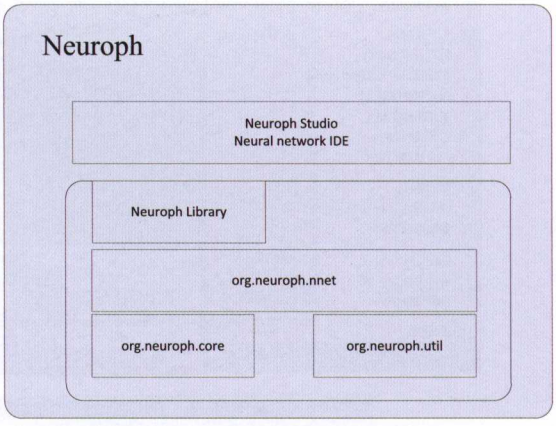
▲图2-4　苹果、香蕉的判定边界



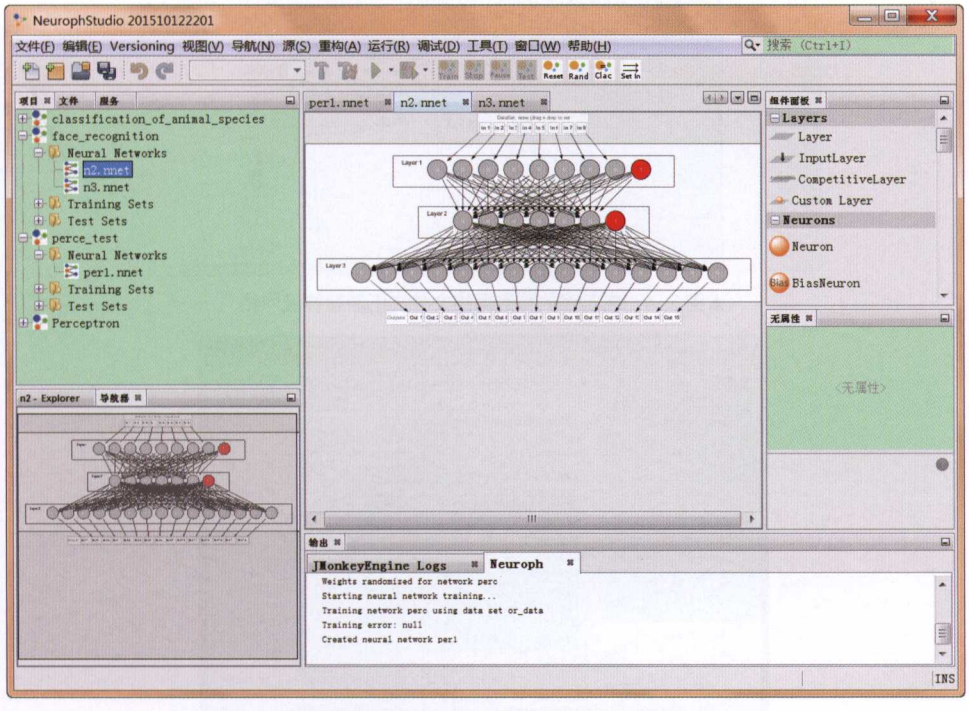
▲图2-5　逻辑与操作



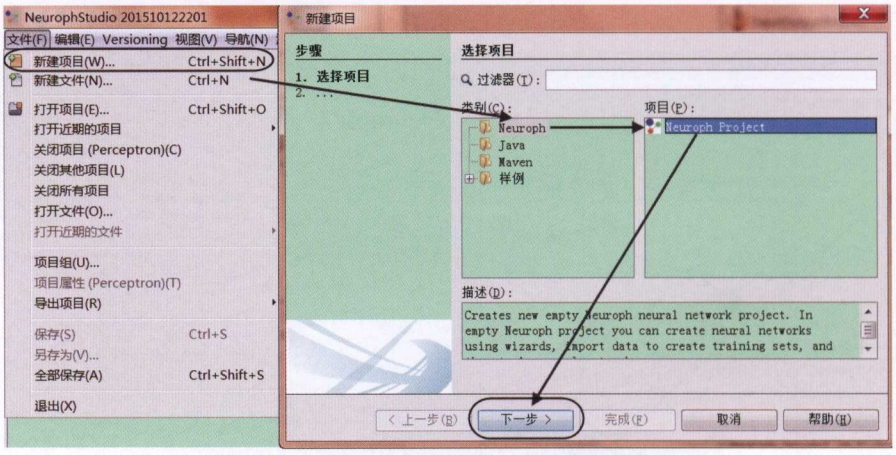
▲图2-6　逻辑与操作的判断边界和权值向量



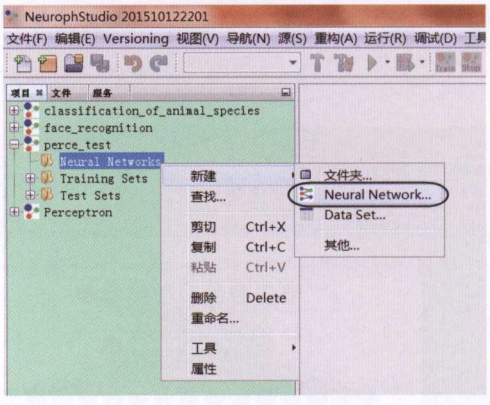
▲图3-1　Neuroph系统架构图



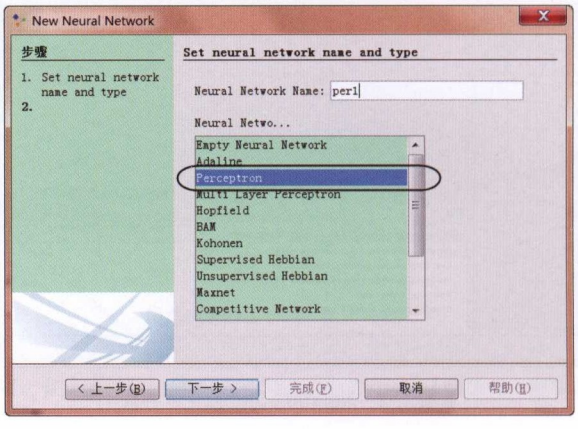
▲图3-2　Neuroph Studio工作界面



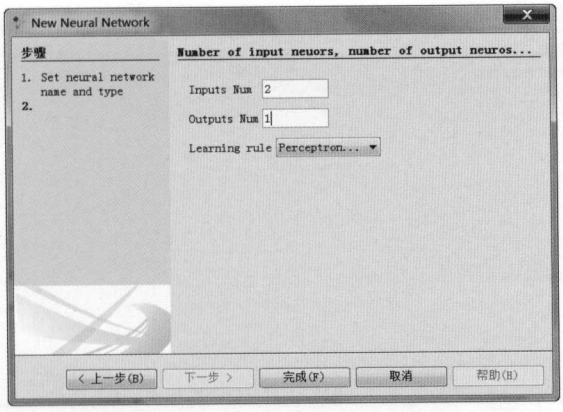
▲图3-3　在Neuroph Studio中新建神经网络工程



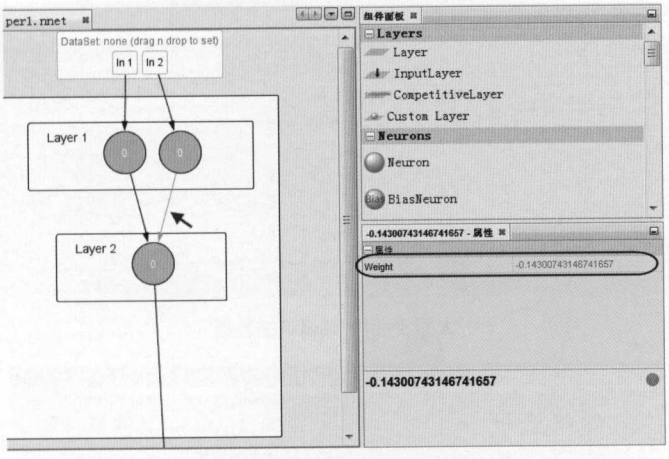
▲图3-4　在工程的神经网络文件夹下新建神经网络



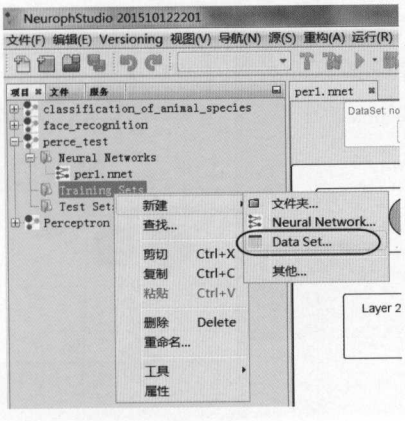
▲图3-5　建立感知机网络



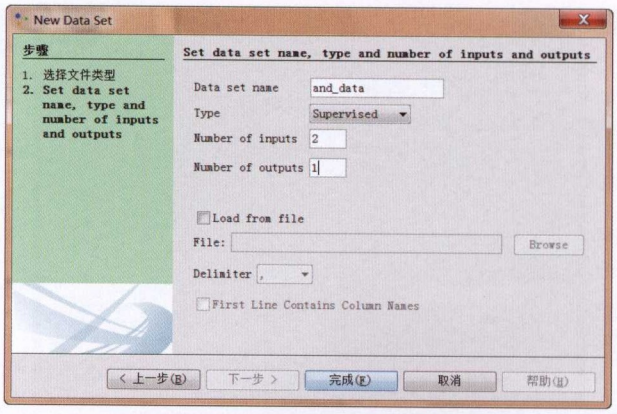
▲图3-6　设置感知机参数



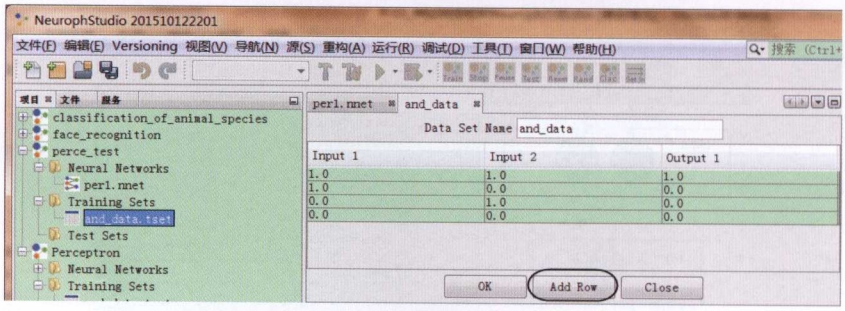
▲图3-8　查看神经元的权重



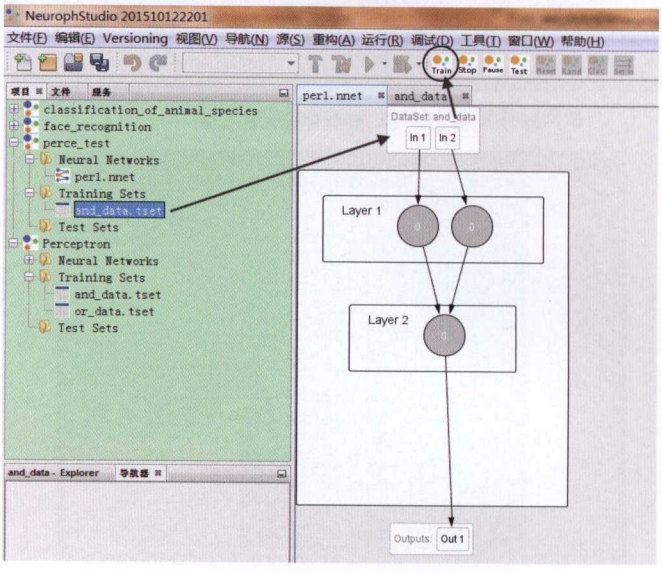
▲图3-9　新建训练数据集



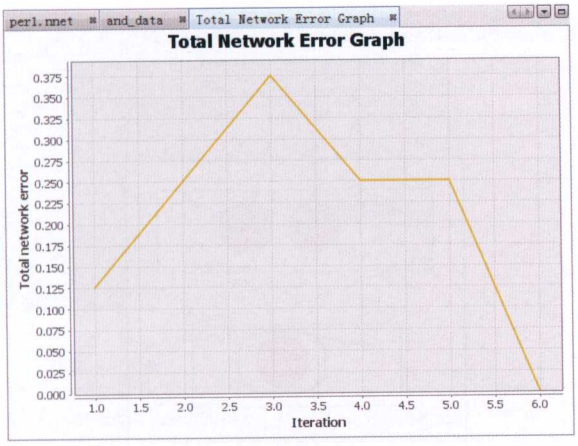
▲图3-10　新建有监督的训练数据集



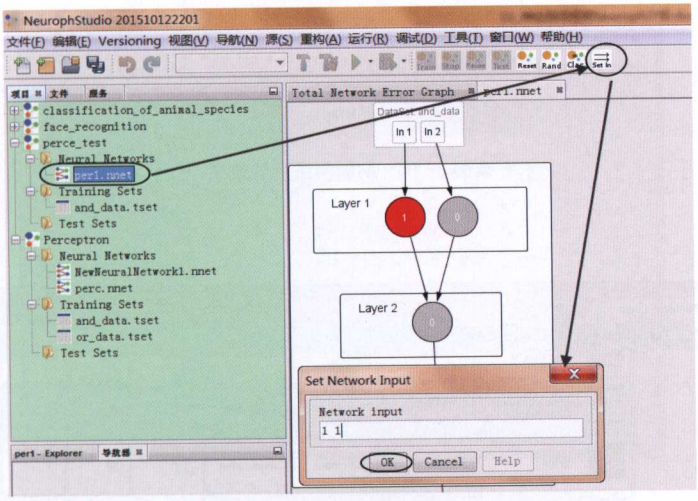
▲图3-11　编辑训练数据



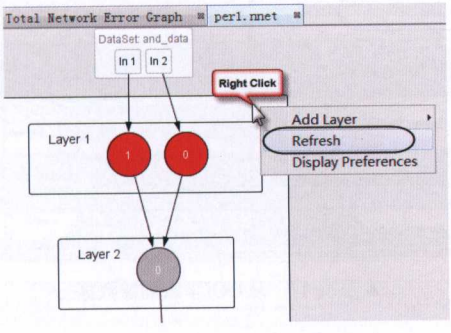
▲图3-12　使用训练数据训练感知机



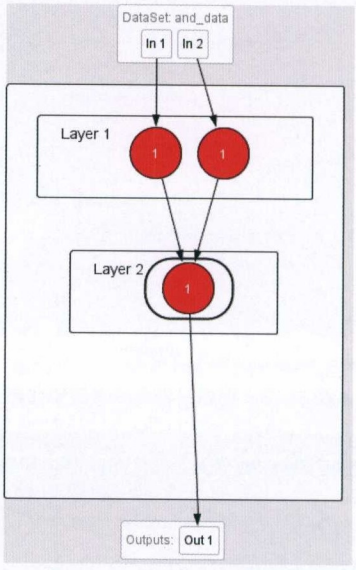
▲图3-13　训练过程中的误差变化图



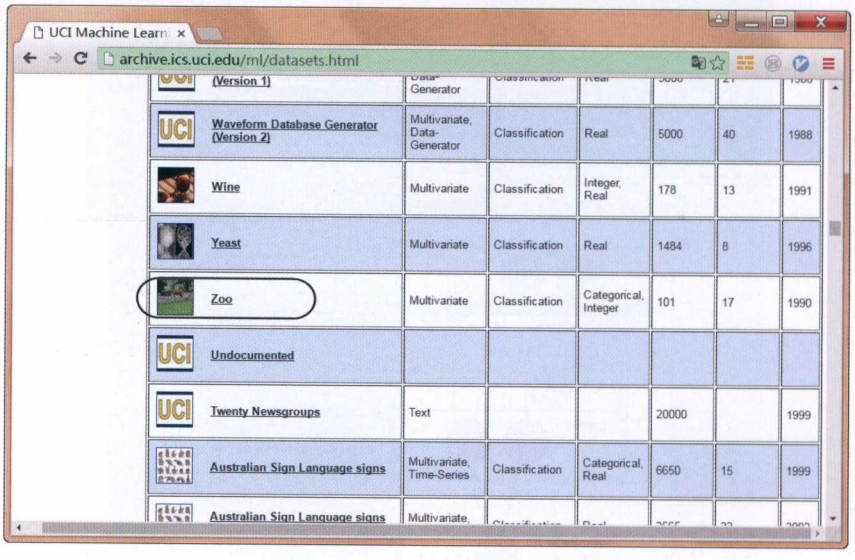
▲图3-14　自定义网络输入



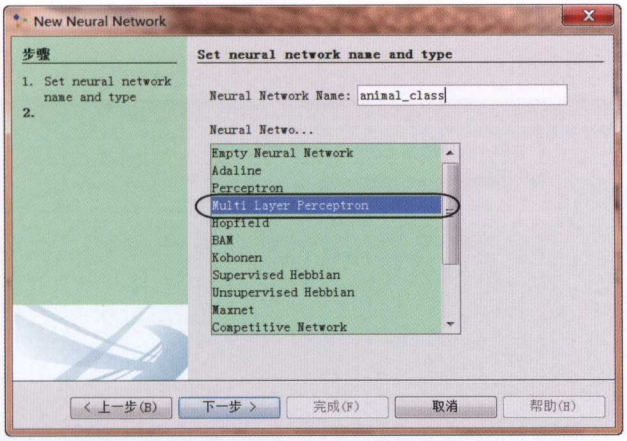
▲图3-15　刷新网络



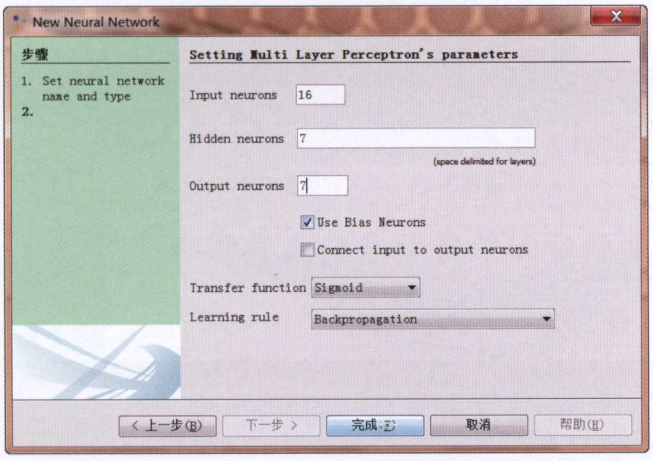
▲图3-16　刷新网络后的效果



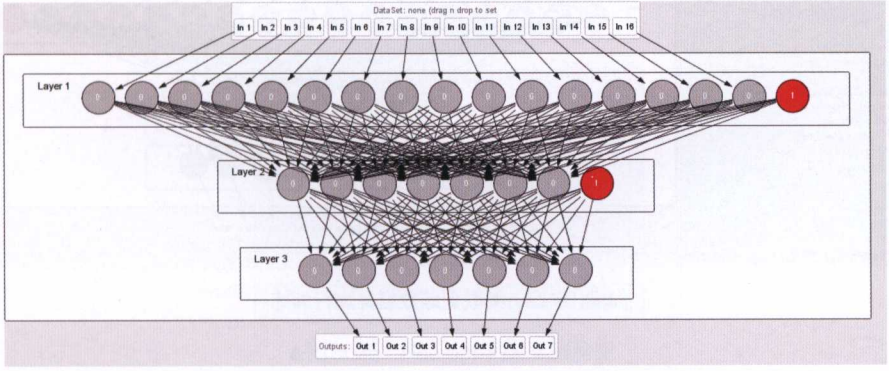
▲图3-17　从UCI下载训练数据



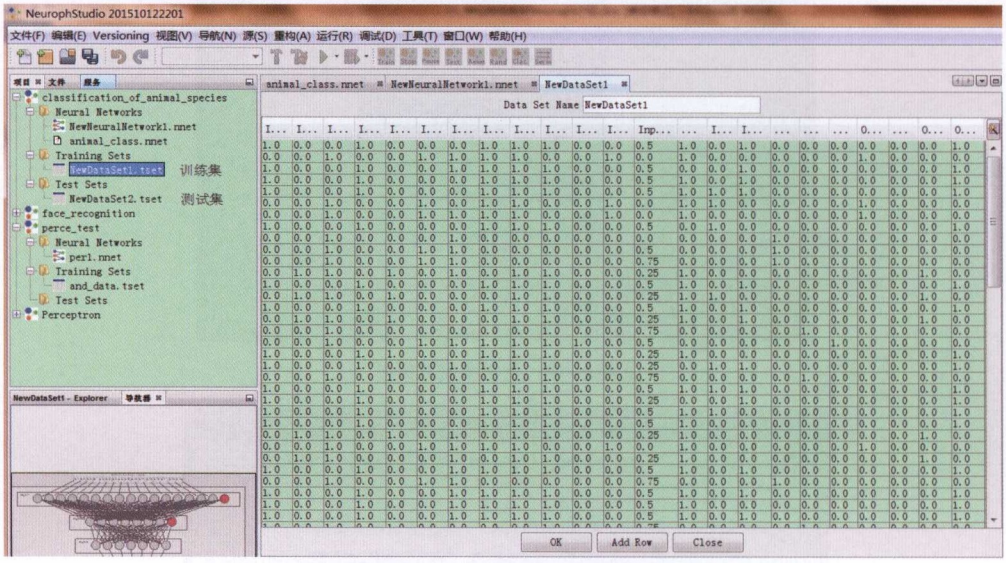
▲图3-18　选择多层感知机



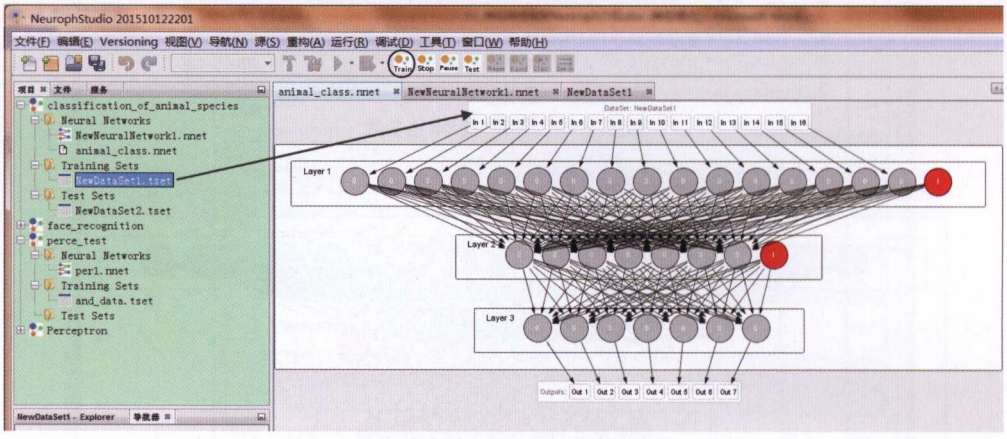
▲图3-19　设置神经网络参数



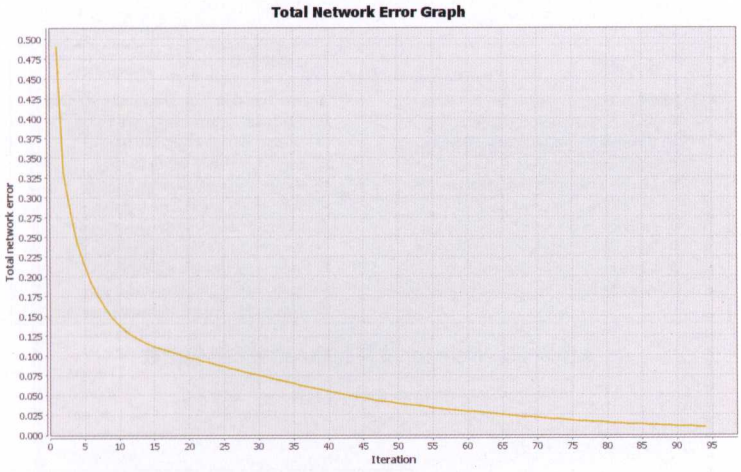
▲图3-20　BP神经网络



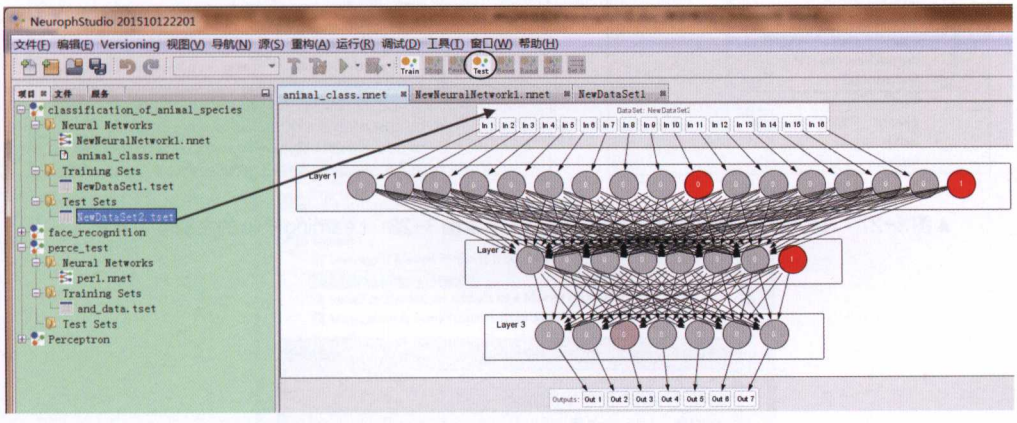
▲图3-21　导入训练数据和测试数据



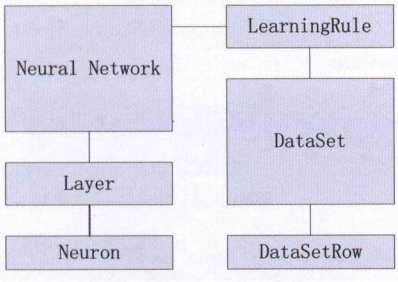
▲图3-22　关联训练数据并进行训练



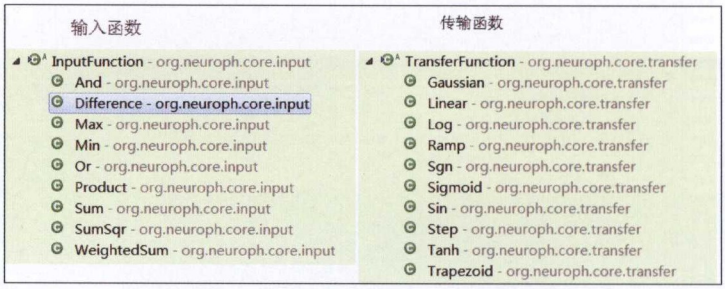
▲图3-23　动物分类整体误差图



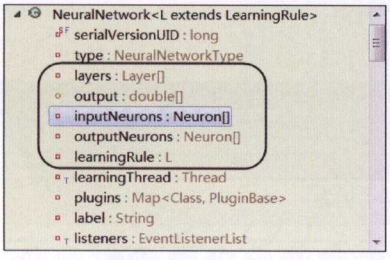
▲图3-24　测试神经网络



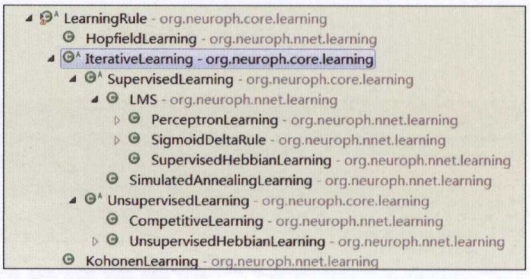
▲图3-25　Neuroph的总体架构图



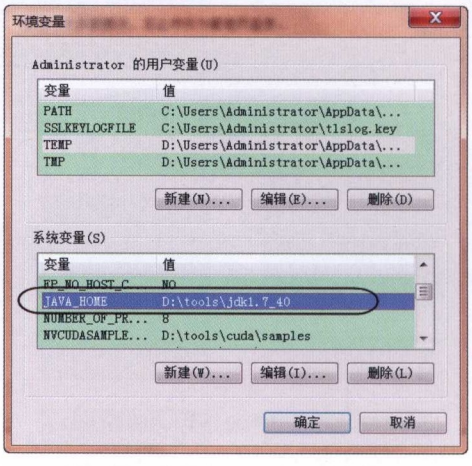
▲图3-26　Neuroph支持的输入函数和传输函数



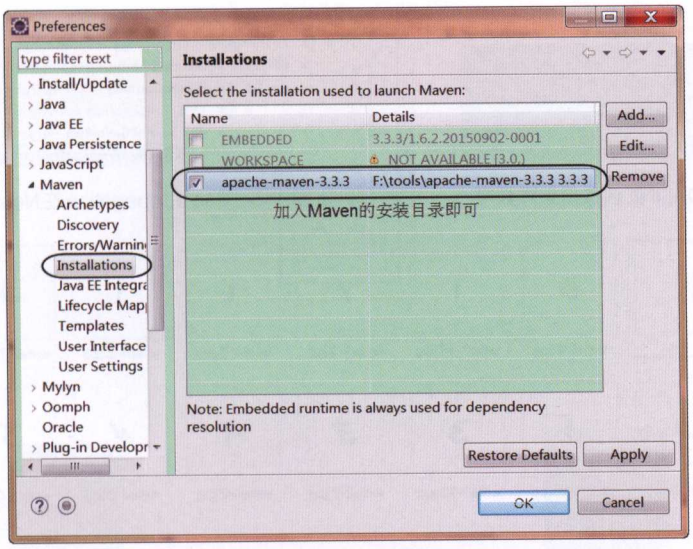
▲图3-27　NeuralNetwork结构



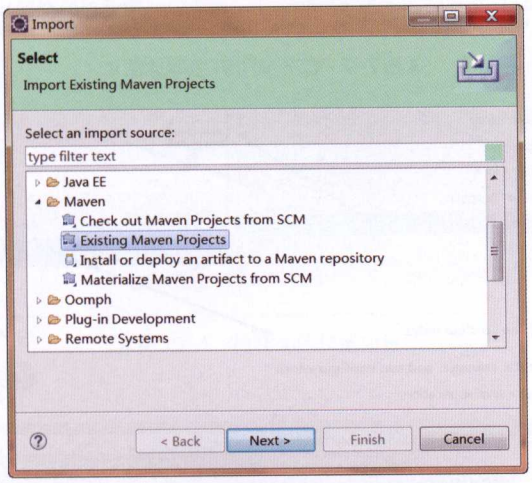
▲图3-28　LearningRule的主要学习算法



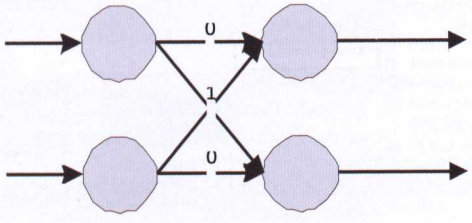
▲图3-29　设置JAVA_HOME环境变量



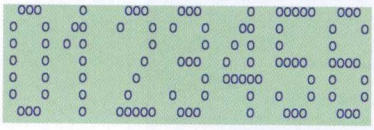
▲图3-30　在Eclipse中配置Maven



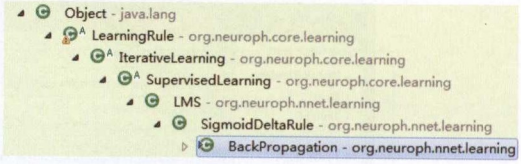
▲图3-31　导入已存在Maven工程



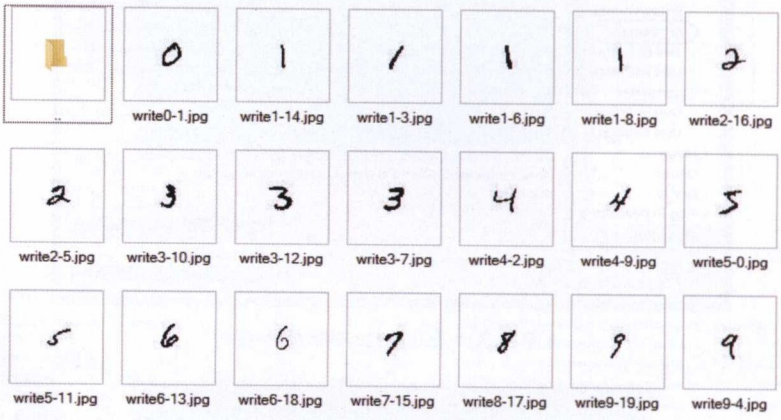
▲图4-2　识别坐标点的象限的感知机



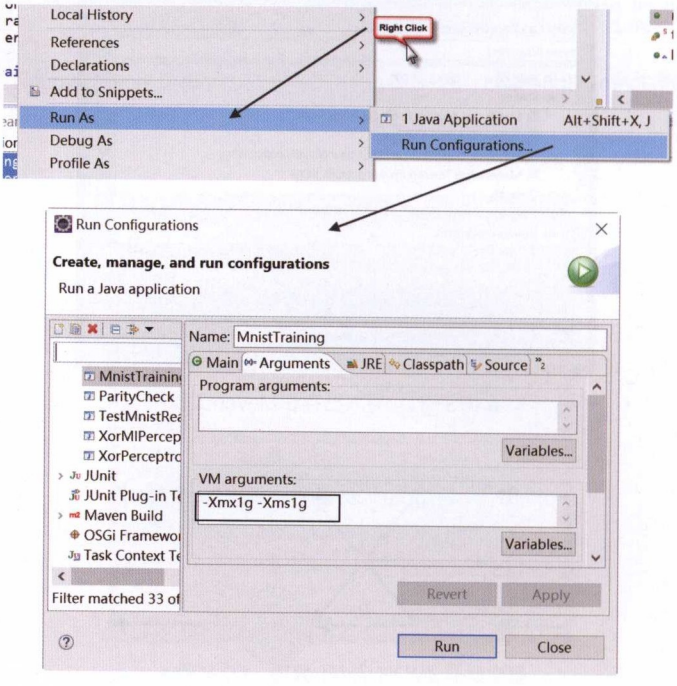
▲图5-3　ADALINE训练用点阵数字



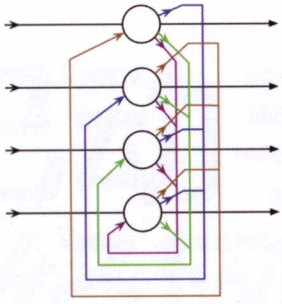
▲图6-6　BackPropagation在Neuroph中的位置



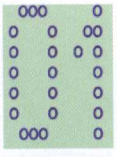
▲图7-7　部分MNIST手写体图片



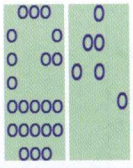
▲图7-8　设置Java的运行时堆空间大小



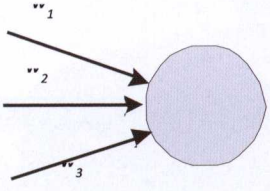
▲图8-1　包含4个神经元的Hopfield网络



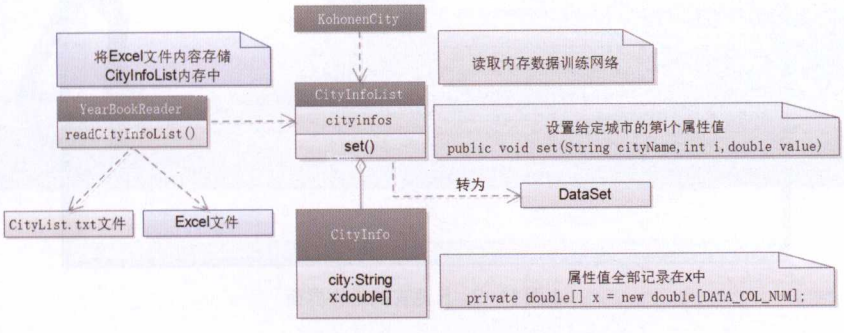
▲图8-3　Hopfield网络训练数字



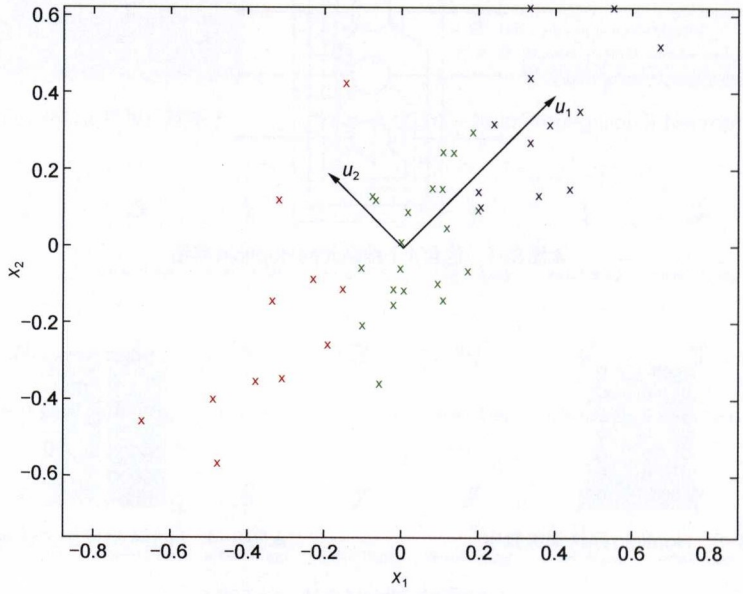
▲图8-4　Hopfield网络测试用数字图像



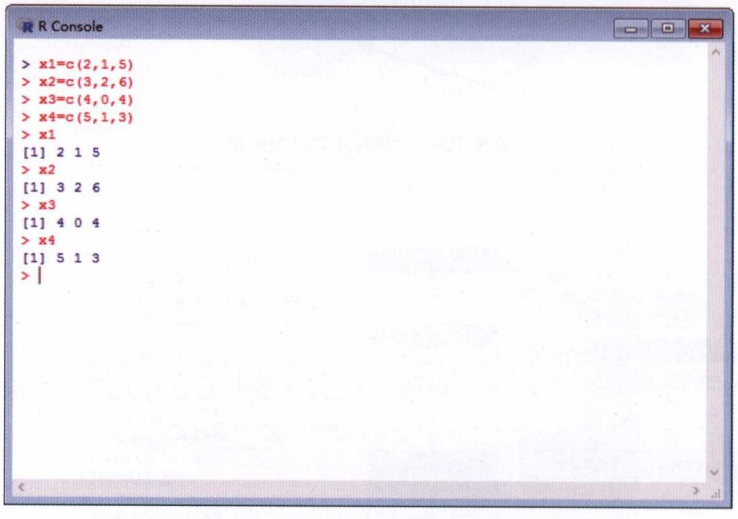
▲图10-1　神经元的权重向量



▲图10-12　城市聚类案例的数据抽取



▲图11-1　PCA方法的几何含义

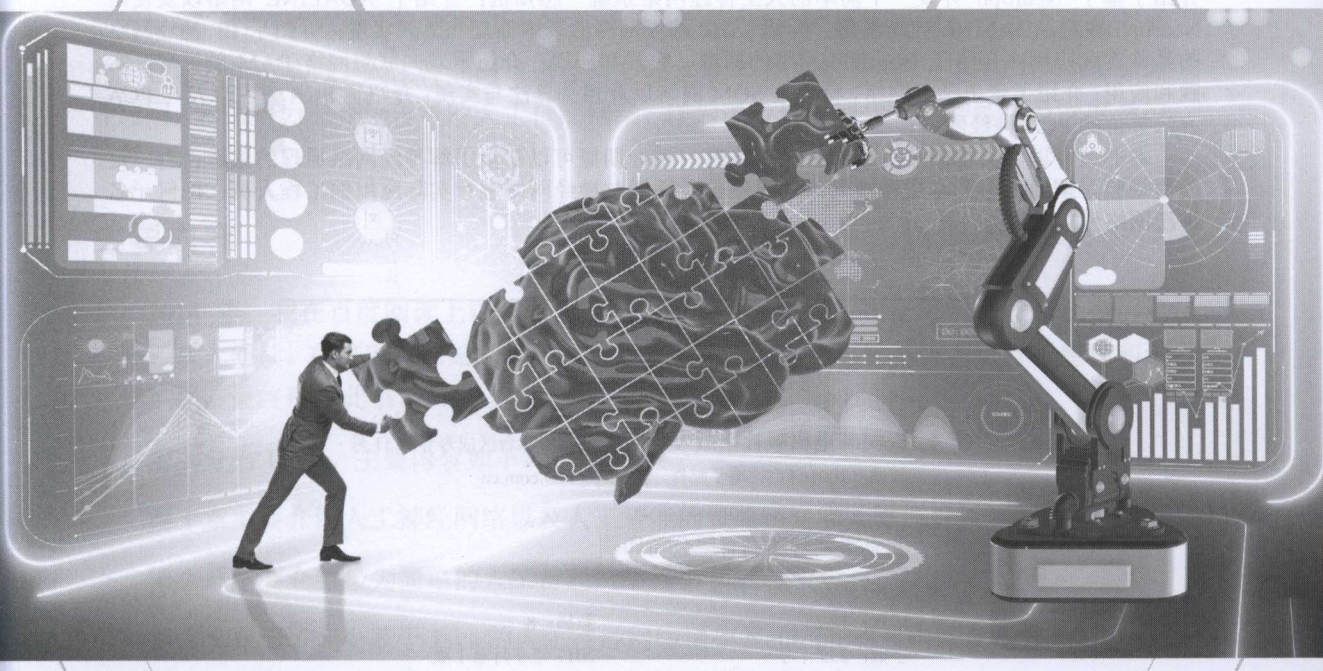


▲图11-2　R语言的工作界面

自己动手写神经网络

葛一鸣◎著

ZIJI DONGSHOU XIE
SHENJING WANGLUO



人民邮电出版社
北京

图书在版编目（CIP）数据

自己动手写神经网络 / 葛一鸣著. -- 北京 : 人民邮电出版社, 2017.9
ISBN 978-7-115-46201-5

Ⅰ. ①自… Ⅱ. ①葛… Ⅲ. ①人工神经元网络－基本知识 Ⅳ. ①TP183

中国版本图书馆CIP数据核字(2017)第182198号

内 容 提 要

本书讲解通俗易懂，使用简单的语言描述人工神经网络的原理，并力求以具体实现与应用为导向，除了理论介绍外，每一章节的应用和实践都有具体的实例实现，让读者达到学以致用。本书分为 11 章，主要内容为：简单的人工神经网络模型和理论应用；介绍了一个基于 Java 的人工神经网络框架 Neuroph；介绍了基于 Neuroph 开发一个简单的人工神经网络系统—感知机；介绍了 ADALINE 网络以及使用 Neuroph 实现 ADALINE 神经网络；介绍了 BP 神经网络的基本原理和具体实现；介绍了 BP 神经网络的具体实践应用；介绍了 Hopfield 网络的原理、实践和应用；介绍了双向联想网络 BAM 的原理、实践和应用；介绍了竞争学习网络，特别是 SOM 网络以及相关算法与实现；介绍了 PCA 方法以及与 PCA 方法等价的 PCA 神经网络。

本书适合以下类型的读者：对神经网络感兴趣，期望可以初步了解神经网络原理的读者；有一定编程经验，期望学习和掌握神经网络的程序员；期望对神经网络进行实际应用的工程人员；任何一名神经网络爱好者。

◆ 著　　葛一鸣
责任编辑　张　涛
责任印制　焦志炜

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路 11 号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京市艺辉印刷有限公司印刷

◆ 开本：800×1000　1/16
印张：12.25　　彩插：8
字数：239 千字　　2017 年 9 月第 1 版
印数：1－2 500 册　　2017 年 9 月北京第 1 次印刷

定价：55.00 元

读者服务热线：(010)81055410　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京东工商广登字 20170147 号

前　言

关于人工神经网络

不可否认，我们已经迎来了人工智能的又一次高潮。与前几次人工智能的飞跃相比，这一次人工智能突破将软件算法、高并发硬件系统以及大数据有机地结合在一起，进而将人工智能推向了最接近人类智能的制高点。位于目前人工智能核心的，也是最炙手可热的算法，被称为深度学习。目前，最为成功的深度学习方法与人工神经网络联系紧密，甚至可以毫不夸张地说，传统的人工神经网络就是深度学习框架的基础。因此，可以认为，目前人工智能以及机器学习的觉醒从本质上说，是以人工神经网络为代表的学习算法和框架的盛行以及广泛实践。人工神经网络已经从实验室里走了出来，得到了更加广泛的实际商业应用。本书致力于介绍传统的人工神经网络，作为读者步入现代人工智能以及深度学习的理论基础。

主要内容

2014 年，我在百度阅读上提交了《自己动手写神经网络》电子书。从发布至今，已有 2 万余人阅读。在整理和总结大部分读者的意见后，我在原书的基础上进行了整理和进一步扩展，最终形成此书。

全书分为 11 章，主要内容如下。

第 1 章主要介绍人工神经网络以及人工智能的发展历史和基本原理。

第 2 章主要介绍最为简单的人工神经网络模型和理论应用。

第 3 章介绍了一个基于 Java 的人工神经网络框架 Neuroph 的架构以及基本使用方法。

第 4 章主要介绍基于 Neuroph 开发一个简单的人工神经网络系统——感知机。

第 5 章介绍了 ADALINE 网络以及使用 Neuroph 实现 ADALINE 神经网络。

第 6 章介绍了 BP 神经网络的基本原理和具体实现。

第 7 章介绍了 BP 神经网络的具体实践应用。

第 8 章介绍了 Hopfield 网络的原理、实践和应用。

第 9 章介绍了双向联想网络 BAM 的原理、实践和应用。

第 10 章介绍了竞争学习网络，特别是 SOM 网络以及相关算法与实现。

第 11 章介绍了 PCA 方法以及与 PCA 方法等价的 PCA 网络。

本书特点

本书的主要特点有。

- 力求通俗易懂，使用尽可能简单的语言描述人工神经网络的原理与理论。
- 力求以具体实现与应用为导向，除了理论介绍外，每章的应用和实践都有具体的实现详解。
- 使用 Java 作为主要语言。与 Matlab 等语言不同的是，Java 语言是目前企业级软件开发最为流行的语言，因此，使用 Java 实现的神经网络具备更强的系统集成能力与实践能力。由于 Java 语言本身通俗易懂，在基本语法上与 C/C++类似，因此，本书也适合没有 Java 基础的程序员。

对读者的要求

本书适合以下类型的读者。

- 对神经网络感兴趣，期望可以初步了解神经网络原理的读者。
- 有一定编程经验，期望学习和掌握神经网络的程序员。
- 期望对神经网络进行实际应用的工程技术人员。
- 任何一名神经网络爱好者。

如果你熟悉 Java，那么通读本书应该不会遇到太大的障碍。但熟悉 Java 并不是阅读本书的必要条件。实际上，只要你有编程经验，并且有面向对象编程的一点概念，那么阅读本书就不会遇到太大的障碍。

本书的资源

本书的每章都提供了相关参考代码，可以帮助读者更好地了解本书内容。有兴趣的读者可以在 http://www.uucode.net/201702/nncode 中下载本书全部代码，本书答疑 QQ 群

424123202。

联系作者

本书的写作过程远比我想象的要艰辛。为了让全书能够更清楚更正确地表达和论述，我经历了好几个不眠之夜。即使现在回想起来，我也忍不住要打个寒颤。但由于我的写作水平和写作时间的限制，书中难免会有不妥之处。为此，读者可以通过邮箱billykinggym@126.com与我联系，也可以通过我的个人网站http://www.uucode.net/与我取得联系。

感谢

本书的出版离不开人民邮电出版社张涛编辑的支持。正是在他的鼓励和帮助下，本书才得以完成，才得以与读者见面。同样，我也要感谢阅读《自己动手写神经网络》电子书的各位读者，正是得到了你们的宝贵意见，我才能更好地完成此书。最后，要感谢我的母亲对我无私的支持和奉献，在这里祝她身体健康，万事如意。

本书编辑联系和投稿邮箱为：zhangtao@ptpress.com.cn。

作者

目　录

第1章　人工神经网络概述

自从人类有文明以来，人类探索的脚步就从未停止过。我们为自己而骄傲，因为我们拥有这颗星球上最高的智慧，足以主宰整个星球；但我们也为自己而迷惑，我们从哪里来，谁创造了我们，并且给予我们与众不同的智慧？对这两个问题，目前还没有答案，但对自身的好奇，迫使我们正在尝试复制自己的智慧。那么，我们能够成为新时代的“上帝”吗？我们可以创造比我们更加聪明的物种吗？一切正在进行中……

1.1　人工智能与神经网络简史

说到人工神经网络，就不得不提人工智能。早在20世纪40年代，世界上第一台计算机ENIAC就问世了。从那时起，一大批科学家开始探讨基于电子元器件制造电子大脑的可能。仅在计算机问世的短短十几年后，1956年，在达特茅斯学院举行的一次会议上正式确立了人工智能的研究领域。在当时，许多人对人工智能充满着美好的幻想，甚至有不少人预言，经过一代人的努力，就可以让机器达到人类的智能。很不幸的是，他们当时太低估人工智能这一领域的难度了。在经过大约20年的发展后，人工智能就遇到了严重的瓶颈，技术的停滞不前使绝大部分人开始认识到这项技术也许并不那么容易，甚至有人已经开始泼人工智能的冷水，认为制造智能机器是一种不可能的事情。随着技术受冷，人工智能研究者也遇到了经费问题。这种情况一直持续到20世纪80年代，专家系统的崛起以及随之带来的大量知识储备的出现，才给了人工智能新的血液。并且非常巧合的是，联结主义学习，也就是神经网络，在这个时期也取得了重大的突破。

但好景不长，那些对专家系统热捧的人也马上失望了，基于大量知识储备和规则的专家系统并不可能实现真正的智能，于是在20世纪80年代末90年代初的时间内，人工智能的发展又遭遇了一次财政危机。一直到1997年的一次重大事件，即IBM公司的“深蓝”

战胜国际象棋世界冠军卡斯帕罗夫，将人工智能再一次展示在人们眼前。2016 年，Google 公司的 AlphaGo 战胜李世石，标志着人工智能又一次回到了鼎盛时期。

1.1.1　人工智能的诞生：1943～1956 年

二战中后期，计算机技术得到了很大的发展。为了配合军方试验，更快地计算导弹轨道、角度等信息，世界上第一台计算机 ENIAC 诞生。这为人工智能的实现提供了很好的技术基础。

神经学的研究显示，人脑而是由神经元构成的电子网络。神经细胞只存在激活和抑制两种状态，没有中间状态。此外，维纳的控制论、香农的信息论、图灵的计算理论相继表明，制造一个基于电子元器件的大脑是可能的。


▲图 1-1　W. S. McCulloch

1943 年，生理学家 W. S. McCulloch 和数学家 W. A. Pitts 提出神经元的数学模型 M-P 模型。这是按照生物神经元的结构和工作原理构造出来的一个抽象和简化了的模型，也就是对一个生物神经元的建模。M-P 模型的名称正是取自两名提出者名字的首字母。图 1-1 所示为提出者之一的 W. S. McCulloch。

1949 年，心理学家 Donald Hebb 在《行为构成》一书中提出 Hebb 算法；他还首先提出了连接主义这一概念。所谓连接主义，是用来形容大脑的工作方式，即大脑的工作是靠脑细胞之间的连接活动完成的。Hebb 指出，如果源和目的神经元均兴奋时，它们之间的突触连接会增强。这也正是 Hebb 算法的生物学基础。Hebb 的最大贡献也在于他提出了有关神经网络工作原理的重要假设：神经网络的信息存储在连接的权值中。图 1-2 所示为 Donald Hebb。

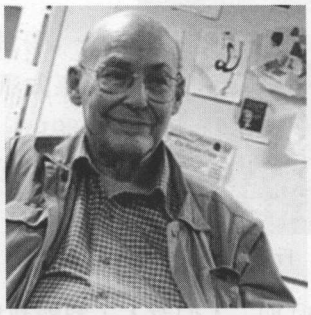
1950 年，图灵发表一篇跨时代的论文，预言了创造具有真正智能的机器的可能性，并且给出了一种用于测试机器是否拥有真正智能的测试方法，即大名鼎鼎的图灵测试。图灵指出：如果一台机器能够与人类展开对话而不能被辨别出其机器身份，那么称这台机器具有智能。图灵测试也是人工智能哲学方面第一个严肃的提案。

1951 年，W. S. McCulloch 和 W. A. Pitts 的学生 Marvin Lee Minsky 与 Dean Edmonds 一起构建了第一台神经网络机（称为 SNARC）。在接下来的 50 年中，Minsky 成为了人工智能的领导人物。图 1-3 所示为 Marvin Lee Minsky。

1955 年，Newell 和（后来荣获诺贝尔奖的）Simon 在 J. C. Shaw 的协助下开发了“逻辑理论家（Logic Theorist）”。这个程序能够证明《数学原理》中前 52 个定理中的 38 个，

其中某些证明比原著中的更加新颖和精巧。Simon 认为他们已经“解决了神秘的心/身问题，解释了物质构成的系统如何获得心灵的性质”(这一断言的哲学立场后来被 John Searle 称为“强人工智能”，即机器可以像人一样具有思想)。

▲图 1-2 Donald Hebb



▲图 1-3 Marvin Lee Minsky

1.1.2 黄金发展期：1956～1974 年

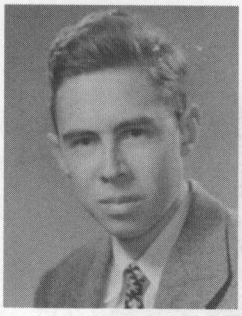
1956 年，Minsky 与 John McCarthy 组织了达特茅斯会议。会议提出的断言之一是“学习或者智能的任何其他特性的每一个方面都应能被精确地加以描述，使得机器可以对其进行模拟”。参会者包括 Ray Solomonoff，Oliver Selfridge，Trenchard More，Arthur Samuel，Newell 和 Simon，他们中的每一位都在后来在 AI 研究的第一个十年中作出了重要贡献。这次会议上，Newell 和 Simon 讨论了“逻辑理论家”，McCarthy 则说服与会者接受“人工智能”一词作为本领域的名称。达特茅斯会议上，人工智能的名称和任务得以确定，同时出现了最初的成就和最早的一批研究者，因此这一会议被广泛认为是人工智能诞生的标志。

达特茅斯会议后的数年，是人工智能的拓荒者时代，几乎每到一处，都能取得令人振奋的成就。人们发现计算机可以解决代数应用题、几何证明题甚至是学习和使用英语。而再此之前，人们从来没有想象过机器竟然可以如此智能。伴随着大量的成就，人们开始对人工智能抱着过分乐观的态度，甚至认为完全智能的机器将在 20 年内出现。

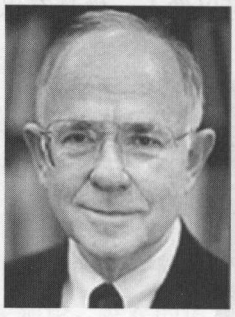
1958 年，计算机科学家 Frank Rosenblatt 提出了第一个真正优秀的人工神经网络——感知机。通过感知机，已经可以解决一些线性可分问题。感知机的理论模型使用了 M-P 模型，并且拥有一套简单可行的学习算法。图 1-4 所示为 Frank Rosenblatt。

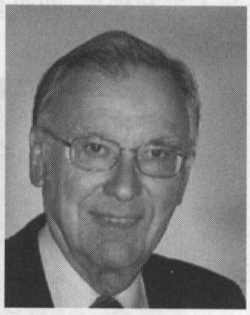
1960 年，电机工程师 Bernard Widrow 和他的研究生 Marcian Hoff 发表《自适应开关电路》，他们用硬件实现了神经网络，提出了 ADALINE 网络，并发表了 Widrow-Hoff 算

法（也就是 LMS 算法）。ADALINE 网络可用于自适应滤波，在本书中大家也将看到如何使用 ADALINE 网络进行印刷体数字识别。图 1-5 所示为 Bernard Widrow 和 Marcian Hoff。



▲图 1-4　Frank Rosenblatt





▲图 1-5　Bernard Widrow 和 Marcian Hoff

在这个时期，人们对人工智能都抱着极大的热情，Minsky 甚至在 1967 年表达过：在一代之内，创造人工智能的问题将在实质上得到解决。

1963 年 6 月，MIT 从新建立的 ARPA（即后来的 DARPA，国防高等研究计划局）获得了 220 万美元经费，用于资助 MAC 工程，其中包括 Minsky 和 McCarthy 五年前建立的人工智能研究组。此后，ARPA 每年提供 300 万美元，直到 20 世纪 70 年代为止。ARPA 还对 Newell 和 Simon 在卡内基梅隆大学的工作组以及斯坦福大学人工智能项目（由 John McCarthy 于 1963 年创建）进行类似的资助。另一个重要的人工智能实验室于 1965 年由 Donald Michie 在爱丁堡大学建立。在接下来的许多年间，这四家研究机构一直是 AI 学术界的研究中心。

1969 年，Minsky 和 S. papert 出版《感知机》一书。在书中，他们仔细分析了以感知机为代表的单层神经网络的功能和局限性，并指出感知机无法处理非线性问题。这本书中对感知机的批评对神经网络的发展造成了极其严重的打击，为后来 10 年神经网络的研究中断埋下了伏笔。

1972 年，Kohonen 提出了 SOM 自组织映射网络。使用该网络，可以通过网络的自学习，实现对数据的聚类。

1972 年也是 Prolog 语言工人的诞生年份。自 1972 年后，Prolog 出现了各种不同的分支。在早期的人工智能研究领域，Prolog 一直是一种主要方法和工具。

1.1.3　第一次低谷期：1974～1980 年

由于人们最初对人工智能的难度进行了错误的判断。到 20 世纪 70 年代，人工智能开

始遭遇各方批评，随之而来的还有研究经费的问题。由于此前人们对人工智能过分乐观，一旦当承诺无法兑现时，对人工智能的资助就大幅减少了，随着人工智能研究经费的取消，到 1974 年，几乎很难找到对人工智能项目的资助了。同时，Minsky 对感知机的批评，对神经网络的发展几乎是毁灭式的，一度导致神经网络销声匿迹了近 10 年。

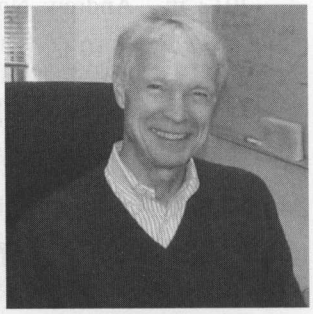
在这一时期，人工智能还饱受一些大学教授的批评。一些哲学家强烈反对人工智能研究者的主张，他们认为哥德尔不完备定理已经证明形式系统（例如计算机程序）不可能判断某些陈述的真理性，但是人类可以。John Searle 于 1980 年提出“中间房间”实验，试图证明程序并不“理解”它所使用的符号。他认为如果机器不理解符号，就不能认为机器在思考。

1.1.4 繁荣期：1980～1987 年

20 世纪 80 年代，一类名为“专家系统”的程序开始被全世界的公司所接受，知识处理成为人工智能领域的焦点。专家系统可以根据一组专业的知识，推演出某一特定领域问题的答案。比如一个名叫 MYCIN 的系统能够诊断血液传染病。专家系统的出现，使得人工智能真正地实用起来。

1981 年，日本经济产业省拨款 8.5 亿美元支持第五代计算机项目，目标是创造能够与人对话、翻译、理解图像并能像人一样思考的机器。其他国家纷纷作出响应，英国和美国也先后对人工智能产业追加投资。这些行为都促成了人工智能的繁荣。

1982 年，John Hopfield 提出了一种新型的神经网络（现在被称为 Hopfield 网络）。这是一种全连接的神经网络，具有自联想和记忆功能，并且易于用硬件实现。这使得沉寂了 10 年之久的神经网络重获新生。图 1-6 所示为 John Hopfield。



▲图 1-6 John Hopfield

1986 年，Rumelhart 和 McCLelland 提出了 BP 神经网络。这是一种按误差逆传播算法训练的多层前馈网络，是目前应用最广泛的神经网络模型之一。

1.1.5 第二次低谷期：1987～1993 年

20 世纪 80 年代，专家系统的发展也接近瓶颈。由于专家系统只能处理特定领域问题，无法形成常识概念，也不可能构建真正的智能。此外，大型专家系统的维护成本

颇高，难以升级，难以使用。于是，人们对人工智能再度失望，并引发了新一轮的财政危机。

1987 年，人工智能硬件市场的需求突然下降，同时苹果和 IBM 的台式机性能也不断提升。到 1987 年时，已经超过了昂贵的 Lisp 机。于是，老系统失去了存在的价值，瞬间土崩瓦解。

1.1.6　再次崛起：1993 年至今

得益于计算机的发展以及摩尔定律，目前，我们有着让前辈们难以想象的计算能力。即便是在人工智能原理尚未取得重大突破的情况下，仅仅依靠计算能力，人们已经在工程上取得了不少瞩目的成绩。1997 年，“深蓝”成为战胜国际象棋世界冠军卡斯帕罗夫的第一个计算机系统。

2005 年，斯坦福大学开发的一台机器人在一条沙漠小路上成功自行行走了 131 公里。

2006 年，Hiton 提出了深度置信网络。这是一种深层次神经网络，是深度学习的前驱。它使用贪心无监督的方法来学习并解决问题，取得了良好的效果。

2009 年，蓝脑计划声称已经成功模拟部分人脑功能。

2012 年，Andrew Ng 与 Jeff Dean 搭建 Google 大脑。Google 大脑使用超过 16000 个 CPU，用以模拟 10 亿个神经元，在语言和图像识别上都取得了突破。

2015 年，微软使用深度学习网络残差学习法，将 ImageNet 的分类错误率降低到 3.57%，这已经低于了人眼识别率。而采用的神经网络深度已经达到 152 层。

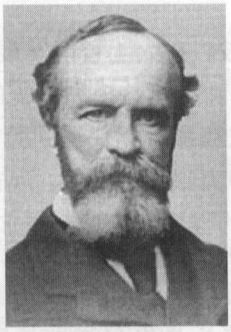
2016 年，拥有 1920 个 CPU 集群和 280 个 GPU 的深度学习系统 AlphaGo 击败李世石，成为第一个不让子击败职业围棋棋手的程序。以“深蓝”为例，“深蓝”比 20 世纪 50 年代的计算机至少要快 1000 万倍。工程上的突破，让人们得以将以前只能停留在理论上的模型实际运行起来并进行测试，这大大加快了人工智能的发展并使之商业化。

1.2　生物学研究对神经网络的影响

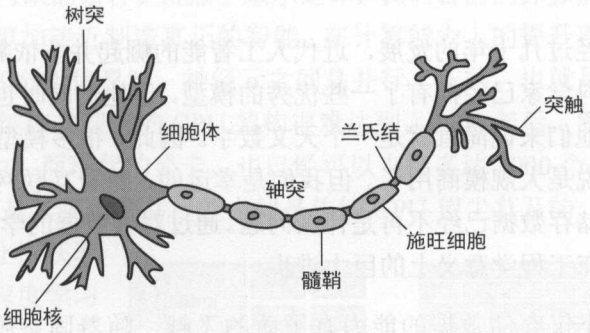
早在 1890 年，William James 在他的《心理学原理》中就详细描述了人脑结构和功能。他指出：当两个基本的脑细胞曾经一起或相继被激活过，其中一个受刺激重新激活时会将

刺激传播到另外一个。图 1-7 所示为 William James。

到 1904 年，生物学家已经基本知晓了神经元的基本结构。图 1-8 所示为一个神经细胞的结构。



▲图 1-7 William James



▲图 1-8 神经细胞的结构

与普通的细胞类似，神经细胞也拥有细胞核和细胞体。与普通细胞不同的是，神经元细胞外围拥有一圈树突的神经末梢。这些树突可以给神经元提供输入，并与其他神经元的轴突相连。树突和轴突相连的点称为突触。这些突触是用来传导电信号的。突触也有强弱之分，表现为对电信号的传导能力，如果传导能力强，那么电信号更容易传递给下一个神经元，反之，电信号就可能会中断。突触的强度应该具有很强的可塑性。根据环境以及自身的历史活动，突触的强度可以发生变化，并且这一过程被认为与学习、成长以及神经系统的成熟密切相关。神经细胞只有一个轴突，轴突很长，可以被认为是神经元细胞的输出通道。轴突的功能是传递细胞的功能电位给突触。单个轴突的直径非常小，在微米级别，但是轴突的长度根据细胞类型不同差异较大。有些神经元细胞的轴突甚至可以长达 1 米。在脊椎动物中，轴突通常会被髓鞘包括，髓鞘可以隔绝电信号干扰，加快动作电位传递，甚至在受伤的情况下，引导轴突再生。轴突末梢与下一个神经元的树突相连，也可能直接连接到下一个神经细胞的细胞体上。

从上面的生物学发现中不难看到，一个神经细胞最为核心的功能是能够根据若干个输入进行电位活动，并将电位信号传递出去，在传递过程中，还可以控制电位信号到下一个神经元的敏感度。神经网络中 M-P 模型正是对这些特征的建模抽象。

人类从大自然学到了很多东西，不少发明创造是大自然给予了我们灵感。在人工神经网络领域更是如此。只有在了解生物学神经系统工作原理的基础上，才更有可能建立一套优秀的人工神经网络。可以说，生物学研究对人工神经网络的发展起到了决定性的引导作

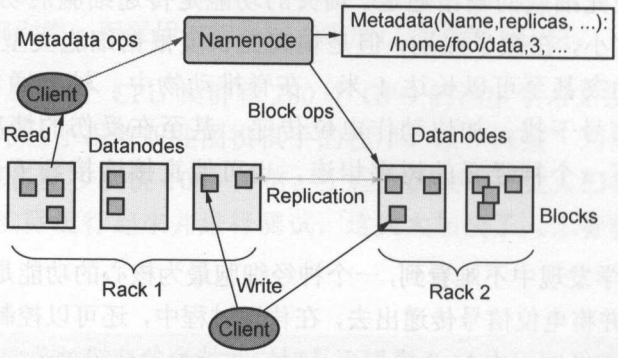
用，也大大促进了人工智能的发展。

1.3 大数据对人工智能的影响

经过几十年的发展，近代人工智能的崛起并非依靠理论突破。在人工智能发展初期，虽然科学家已经拥有了一些优秀的模型，但他们同时也发现，要训练这些模型所需的数据量对他们来说简直就是一个天文数字。因此，很多模型根本无法在那个年代得到验证，更不用说是大规模商用了。但我们是幸运的，随着互联网的崛起和大数据的诞生，现在获取以及储存数据已经不再是什么问题。通过对大数据的学习，我们的程序就能变得更加智能。这是在工程学意义上的巨大进步。

不仅存储数据的能力有了质的飞越，随着固态硬盘的普及，读取数据的速度也有了数量级的提升。这意味着智能软件可以在很短时间内接触到这些数据，大大缩短训练时间。

图 1-9 所示显示了 Hadoop 分布式文件系统的架构。基于该文件系统，可以将大量数据分散存储在多个不同的数据节点上，这极大地增加了单节点的存储能力，为智能软件的训练提供了良好的数据基础。其中 Namenode 存储着文件系统的索引和元数据，Datanode 存储着具体的数据文件。整个 Hadoop 系统中，Datanode 的数量可以多达上千台，其存储能力可想而知。



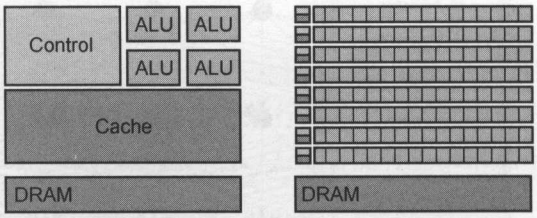
▲图 1-9　Hadoop 分布式文件系统的架构

为访问这些海量数据，一批优秀的大数据计算模型也应运而生，典型的代表有 MapReduce，Spark，Storm 等。这些计算模型为大数据的分析提供了算法基础。

1.4 计算机硬件发展对人工智能的影响

计算能力是发展人工智能必不可缺的条件。得益于摩尔定律，我们目前的计算能力是我们前辈的几千万倍。这使得我们更加接近制造真正的智能。在计算能力上的提升不仅仅是在 CPU 层面。生物大脑在处理和分析信息时，神经元之间是并行工作的。也就是说我们的大脑拥有极高的并发度。这单纯依靠现代的 CPU 架构很难达到。2000 年后，图形处理器 GPU 已经具备了可编程的能力。而现代的显卡，也已经可以支持多达 3000 个核心，虽然 GPU 不及 CPU 那么强大，但是其强大的并行能力却是传统 CPU 望尘莫及的。计算能力的提升使得我们有能力深入分析获取的海量数据，而这一切也正是促使深度学习蓬勃发展的硬件基础。从目前深度学习的发展趋势上看，基于 GPU 的计算显然会成为未来的趋势。

图 1-10 所示简单展示了 CPU 和 GPU 的架构差别。



▲图 1-10 CPU 和 GPU 的架构差别

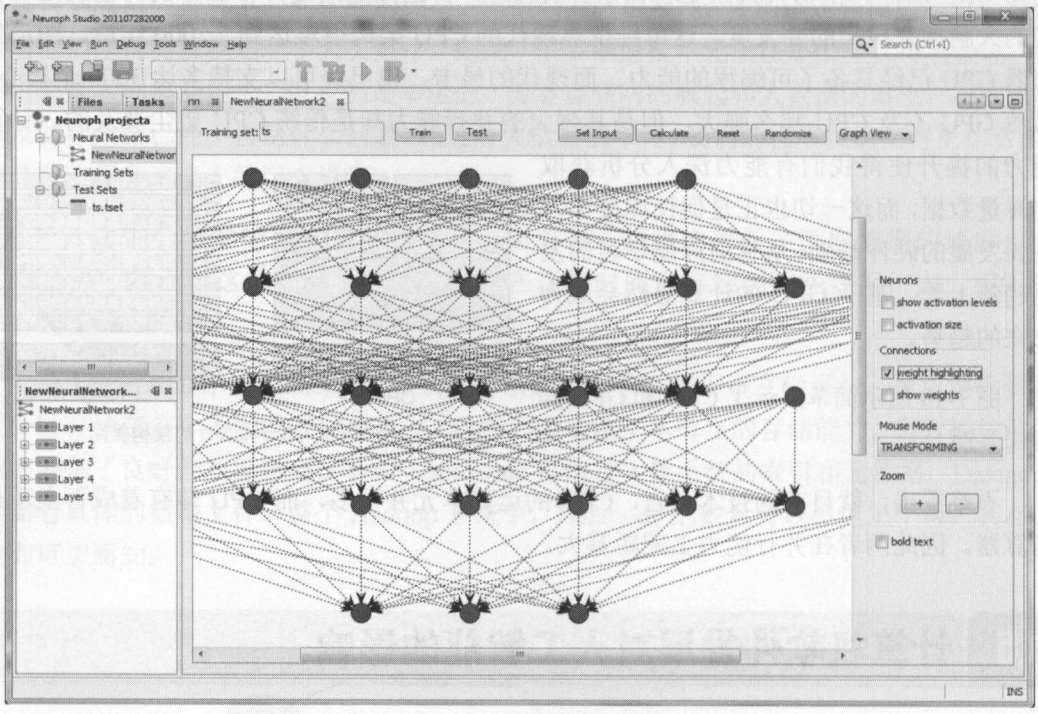
不难看出，就目前的技术而言，CPU 的运算单元并不多，而 GPU 却有着成百上千个运算器。因此两者在并行能力上差距甚大。

1.5 计算机软件发展对人工智能的影响

人工智能作为一种软件实现，其本质是一段程序。软件以及软件工程的发展对人工智能也有着直接的促进作用。通常，一个成熟或者商用的智能程序，不会是一个“小”软件，而对大型软件的维护和开发需要更好的工具。显然，Java 或者更广泛地说是 Java 平台，就成为开发人工智能程序的不错选择。借助这种功能强大且有着良好社区支持的开发平台，不仅大大降低了智能软件的开发和维护成本，同时，由于 Java 社区固有的开源的特点，大量优秀的人工智能框架也孕育而生。比如，本书中提及的 Neuroph 框架就是典型的案例。此外，还有基于大数据的运行平台 Apache Mahout、基于 Spark 的智能算法库 Spark MLLib、开源深度学习系统 Deeplearning4j，这些都是基于 Java 平台的人工智能软件。除 Java 社区外，Google 最新的开源智能系统 TensorFlow、开源深度学习框架 Caffe 都是该领域杰出的项目。可以说，随着软件事业的发展，人工智能系统正处于百花齐放的阶段，各

种优秀项目层出不穷，大大促进了人工智能系统的普及与商用。可以毫不夸张地说，目前正是人工智能软件前所未有的普及阶段，其涉及人群、关乎领域与过去十几年相比有着天壤之别。同时，得益于大数据系统以及硬件能力的提升，可以预测人工智能未来将有着更加广泛的实际应用。

图 1-11 所示展示了使用 Neuroph 人工神经网络 IDE 构造神经网络的界面。



▲图 1-11　Neuroph 人工神经网络 IDE 工具箱

1.6 人工智能的广泛应用

人工智能发展到现在，已经有了极其广泛的应用场景。除了在一些专业领域中的专家系统外，在日常生活中也出现大量人工智能的应用，比如大家熟知的 AppleSiri。Apple Siri 是一款内置在 Apple iOS 系统中的人工智能助理软件。基于自然语言处理技术，用户可以使用自然语言与手机进行交互，完成数据搜索、手机设置等众多服务。

在 Windows 10 中，新的 Hello 人脸识别系统可以帮计算机“认识”自己的主人，而

免去输入用户名密码的步骤。此外，类似的人脸识别系统也广泛应用于摄像健康系统、用于识别恐怖分子。百度云还使用该技术用于管理用户照片，帮助用户对海量照片进行人物筛选。

在公共交通出行上，Google 和百度以及大型汽车生产商都已经在研发自动驾驶系统，以解放驾驶员的双手。特斯拉的自动驾驶系统已经商用，开启了无人驾驶的新时代。可以预计，在不远的将来，自动驾驶系统必将普及而影响到我们每一个人。

在信息安全领域，入侵检测早已经不是新鲜事，基于大数据的入侵检测甚至是自我修复系统已经在阿里云商用，用以确保千千万万的网站的数据安全。在美国拉斯维加斯举办的机器黑客大赛，甚至一改由人进行网络攻防的形式，而换由机器进行网络攻防、漏洞扫描，大有取代甚至超越人类黑客的趋势。

在医疗领域，百度依托大数据和云平台，发布了“百度医疗大脑”。“百度医疗大脑”可以通过人工智能协助医生更高效地服务病人，甚至可以与用户进行多轮交流，依据用户症状，提出问题，反复验证，给出最终建议。从未来发展趋势看，人工智能有望未来几年在医疗领域发挥重要作用。

展望未来，目前蓬勃发展的物联网和智能家居最终也必然会与人工智能相结合。而这将成为一个颠覆性的产业。将来，我们的门窗、家电、浴室都能够知道我们需要什么，了解我们的生活习惯，并自动为我们服务。

可以看到，随着人工智能的不断发展，人工智能已经开始逐步渗透到日常生活的各方各面，最终将重塑整个社会形态，给我们一个完全不同的世界。

第2章　人工神经元模型与感知机

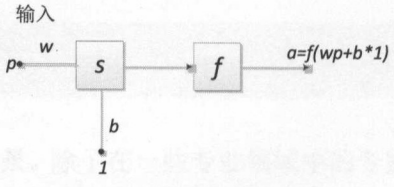
如同一个简单的生物体可以只包含一个细胞一样，最简单的人工神经网络只需要由一个神经元构成。本章将介绍这种最简单的神经网络感知机。在这里，你可以知道：

- 人工神经元的基本结构以及 M-P 模型；
- 感知机的概念以及工作原理；
- 如何构造一个感知机处理二分类问题。

2.1　人工神经元组成要素

2.1.1　人工神经元的基本结构

人工神经网络的基本单元是人工神经元。这里先简单介绍一下神经元的基本结构，也就是 M-P 模型。该模型是神经网络的基础，于 1943 年由生理学家 W. S. McCulloch 和数学家 W. A. Pitts 提出。图 2-1 所示即为该模型。其中，p 是神经元的一个输入，w 是该输入 p 到神经元的权值，也就是连接的强度（类比突触），p 可以是其他神经元对该神经元的输入，也可以是外部系统对该神经元的刺激输入。b 是神经元的偏置（更准确地可以理解为偏置的权重），1 是神经元的固定偏置输入，偏置用来模拟生物神经元的内在化学性质。s 是累加器，也可以叫作神经元的输入函数，通常它对 p 和偏置进行加权求和运算，s 的输出为 $wp+b*1$，s 的输出通常成为净输入 net。偏置 1 和 b 并不是神经元必需的，实际上在很多神经网络中可以没有这个部分。s 的输出将进入 f，f 为神经元的传输函

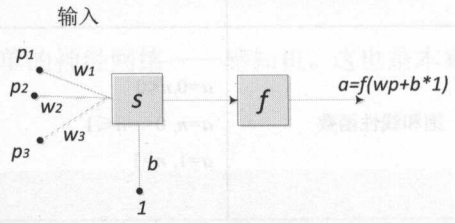


▲图 2-1　单输入的神经元模型

数，$a=f(net)$即为这个神经元的输出，a 也就是模拟了生物神经元的轴突信号。

在该模型中，w 和 b 是可以调整的参数，传输函数 f 和输入函数 s 可以根据需要进行选择，但是在确定后一般不再更改。

在绝大部分场合，神经元的输入往往并不是单一的，因为单一的输入意味着该神经元只能与外接单一的其他神经元相连接，这显然是不够的。因此，为了使得神经元可以连接多个不同的上游神经元，需要让神经元可以接受多个不同的输入 p。



▲图 2-2 多输入的神经元模型

一个通用的神经元必须是可以接受多个输入的。图 2-2 所示的模型中，神经元拥有 3 个输入 $p1$，$p2$ 和 $p3$。其中，$w1$，$w2$，$w3$ 和 b 是根据网络情况不断进行调整的，而传入函数 s 和传输函数 f 是事先选定的，那究竟有哪些函数可以选择呢？传入函数比较简单，最常用的只有按照权重求和（但也存在其他形式）。在本例中，s 处的输出（即神经元净输入 n）就是：

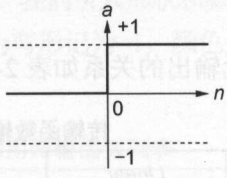
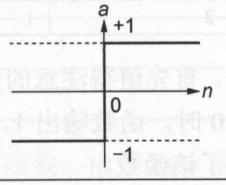
$$n=p1*w1+p2*w2+p3*w3+b*1$$

该输出将作为参数传入传输函数 f，传输函数 f 的输出将作为该神经元的最终输出。

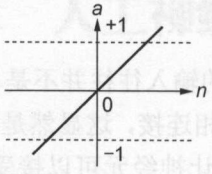
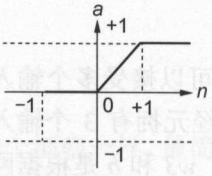
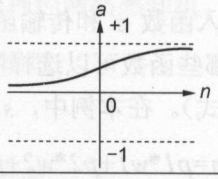
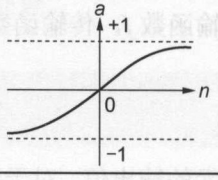
2.1.2 传输函数类型

传输函数直接决定了神经元的输出值，对于整个神经元来说至关重要。选择不同的神经元将使得神经网络具有不同的特性。一般来说，常用的传输函数有表 2-1 所列几种。

表 2-1 常用传输函数

函数名称	映射关系	图像	缩写	说明
阶梯函数	$a=0, n\leqslant 0$ $a=1, n>0$		*Step*	n 大于等于 0 时，输出 1，否则输出 0
符号函数	$a=-1, n\leqslant 0$ $a=1, n>0$		*Sgn*	n 大于等于 0 时，输出 1，否则输出-1

续表

函数名称	映射关系	图像	缩写	说明
线性函数	$a=n$		*Linear*	n 本身就是神经元的输出
饱和线性函数	$a=0, n<0$ $a=n, 0<=n\leqslant 1$ $a=1, n>1$		*Ramp*	n 小于 0 时，输出 0；n 在 0 到 1 区间时，输出 n；n 大于 1 时，输出 1
对数 S 形函数	$a=1/(1+exp(-n))$		*Sigmoid*	有界函数，无论 n 如何，输出永远在(0,1)的开区间
双曲正切 S 形函数	$a=\dfrac{exp(n)-exp(-n)}{exp(n)+exp(-n)}$		*Tanh*	有界函数，无论 n 如何，输出永远在(−1,1)的开区间

在图 2-2 中，假设 $p1=1$，$p2=0$，$p3=2$，$w1=1$，$w2=-1$，$w3=1$，$b=-1$，则神经元的净输入为：$p1*w1+p2*w2+p3*w3+b*1$

$=1*1+0*-1+2*1-1$

$=2$

此时，传输函数与神经元输出的关系如表 2-2 所列。

表 2-2　　传输函数输出值

Step	*Sgn*	*Linear*	*Ramp*	*Sigmoid*	*Tanh*
1	1	2	1	0.881	0.964

这里再强调三种传输函数。首先值得注意的是 *Step* 函数，它非常简单，当输入小于 0 时，函数输出 0；当输入大于 0 时，函数输出 1。该函数可以把输入简单地分为两类。在后续讲到的感知机中，就使用了该函数。

其次值得注意的函数是*Linear*线性函数，它总是简单地返回输入值。在一个ADALINE网络中，会使用该函数。ADALINE类似于感知机，但是因为使用线性函数和其对应的改良学习算法，所以ADALINE比感知机可以更好地处理网络噪声。

最后值得注意的函数是*Sigmoid*函数，它接受任意实数输入，并将结果对应到0和1之间。该函数是可导的。因此，在BP神经网络中使用该函数（BP神经网络学习过程中，需要对传输函数进行求导）。

单个神经元看似简单，但也足可构成一个最简单的神经网络——感知机。这也是本章后半部分的重点内容。

2.2 感知机

感知机可以处理简单的线性分类问题。例如，现在有两类水果，苹果和香蕉，人们通过苹果和香蕉的形状和颜色差别，来区分苹果和香蕉两种水果。刚出生的婴儿无法区分苹果和香蕉，因为在他们的大脑里，没有对应的分类信息。但通过不断的训练和外部刺激，告诉他们红色的圆形的是苹果，黄色的弯形的是香蕉，不需要多久，婴儿就可以分区这两类水果。用类似的方法也可以让感知机正确地对苹果和香蕉进行区分。

2.2.1 使用感知机识别水果

最简单的单层感知机可以只由一个神经元构成。在单层神经元感知机中，网络接受若干输入，并通过输入函数、传输函数给出一个网络的输出。这个网络已经可以解决苹果和香蕉的区分问题。在本节中，将具体介绍其内部原理。

首先，我们确定感知机的输入。在此，我们引入形状和颜色两个变量，苹果的形状为圆形记为1，颜色为红色记为1；香蕉的形状为弯形记为−1，颜色为黄色记为−1。则有输入p如表2-3所列。

表2-3 常用传输函数列表

p	形状	颜色
苹果	1	1
香蕉	−1	−1

其次，可以确定传输函数f为*Step*函数。由于*Step*函数只能有1和0两种输出，所

以在此定义输出 1 表示苹果，输出 0 表示香蕉。

令权重 $w1$=1，$w2$=1，b=0，则图 2-3 所示的感知机有两个输入，并且都是 1，表示圆形并且颜色为红色。

此时：

$$net=p1*w1+p2*w2+b$$
$$=1+1+0$$
$$=2$$
$$step(net)=1$$

可以看到，当输入苹果属性（形状 1，颜色 1）时，感知机正确输出了 1，表示苹果。

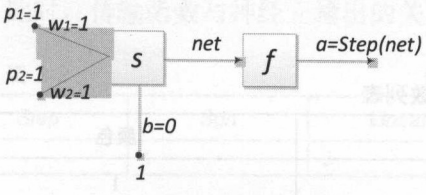
同样道理，当输入−1、−1 的香蕉属性时，感知机反应如下：

$$net=p1*w1+p2*w2+b$$
$$=-1-1+0$$
$$=-2$$
$$step(net)=0$$

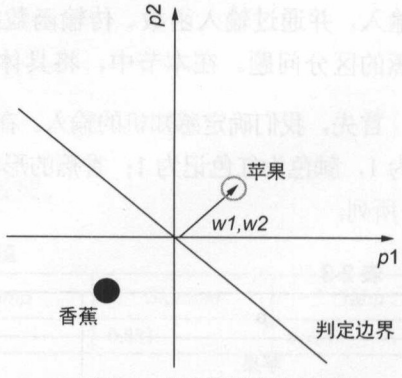
感知机又做出了正确的判断，当遇到弯的黄色水果时，感知机判断其为香蕉。

由于在这里使用 *Step* 函数作为传输函数，因此净输入 *net* 为 0 的 $w1$ 和 $w2$ 构成了这个感知机的判定边界。在判定边界上，净输入为 0；在判定边界的两侧，则分别大于 0 和小于 0。

此处：b=0,$w1$=1,$w2$=1。由此得到判定边界：$p1+p2$=0，如图 2-4 所示。



▲图 2-3　感知机识别苹果



▲图 2-4　苹果香蕉的判定边界

这里有一个重要特性：权值向量($w1$,$w2$)和判定边界是垂直的。这是因为权值向量($w1$ 和 $w2$ 构成的向量)和输入向量($p1$ 和 $p2$ 构成的向量)的内积等于−b(此处 b 为 0)。这意味着输入向

量在权值向量上的投影为$-b/|\boldsymbol{w}|$，在本例中均为0。而只有当判定边界和权值向量垂直时，才能满足此条件。

根据这个特点，就可以通过作图法快速求解出权值向量(*w1*,*w2*)，并用以构建感知机网络。

2.2.2 让感知机记忆逻辑与

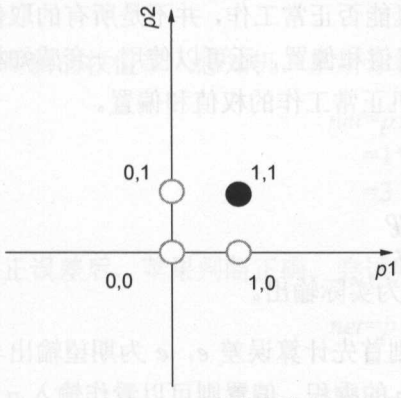
上一小节，我们构造了一个简单的感知机网络，并让它识别苹果和香蕉。基于这个简单的案例，提出了判断边界和权值向量之间的关系。在本节中，我们将基于这种关系，快速求解权值向量。本节中，我们将使用这个方法构造一个可以记忆逻辑与的感知机。

逻辑与运算是一个常用的逻辑运算，其定义如下：

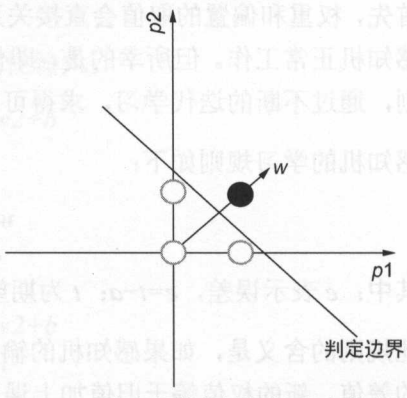
$$1 \ and \ 1 = 1$$
$$1 \ and \ 0 = 0$$
$$0 \ and \ 1 = 0$$
$$0 \ and \ 0 = 0$$

逻辑与的操作如图2-5所示。

显然，对于输入*p1*和*p2*，其结果被明显分为0和1两类。图2-5中，空心点表示0，实心点表示1。我们可以通过一条判断边界，将其分为两类，并且很容易画出一条权值向量与这个判断边界垂直，如图2-6所示，任取判断边界为*p1*+*p2*=1.5。该边界可以将四种情况正确分为两类。权值向量$\boldsymbol{w}$与判断边界垂直，这里可以简单地取(1,1)。



▲图2-5 逻辑与操作

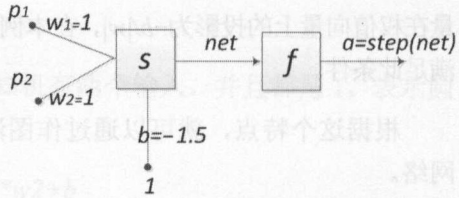


▲图2-6 逻辑与操作的判断边界和权值向量

因此，在判断边界上任取一点(1.5,0)代入*w1*p1*+ *w2*p1*+*b*=0，得到：

$1.5+b=0$，即 $b=-1.5$。

至此，根据这条判断边界，我们求得了一个可用的 $\boldsymbol{w}$ 与 b，得到的感知机如图 2-7 所示。



▲图 2-7　可以记忆逻辑与操作的感知机

下面代入数据进行验算：

当 $p1=0, p2=0$ 时，

$$step(w1*p1+w2*p1+b)=step\ (0*1+0*1-1.5)=step(-1.5)=0$$

当 $p1=1, p2=0$ 时，

$$step(w1*p1+w2*p1+b)=step\ (1*1+0*1-1.5)=step(-0.5)=0$$

当 $p1=0, p2=1$ 时，

$$step(w1*p1+w2*p1+b)=step\ (0*1+1*1-1.5)=step(-0.5)=0$$

当 $p1=1, p2=1$ 时，

$$step(w1*p1+w2*p1+b)=step\ (1*1+1*1-1.5)=step(0.5)=1$$

可以看到，感知机对于所有的输入都给出了正确的答案。

2.2.3　感知机的学习算法

虽然感知机在此时已经可以做出正确的判断，但是读者一定会觉得很疑惑，权值 $\boldsymbol{w}$ 和偏置 b 一定需要人为设计吗，$\boldsymbol{w}$ 的取值又为什么得取(1,1)呢？如果取其他值，感知机还可以正常工作吗？是否存在一种方法可以让感知机自己学习到合理的权重和偏置，而不要实现设计呢？

首先，权重和偏置的取值会直接关系到感知机能否正常工作，并不是所有的取值都可以使感知机正常工作。但所幸的是，即使弄错了权值和偏置，还可以使用一套感知机的学习规则，通过不断的迭代学习，求得可以使感知机正常工作的权值和偏置。

感知机的学习规则如下：

$$w_{new}=w_{old}+ep$$
$$b_{new}=b_{old}+e$$

其中：e 表示误差，$e=t-a$；t 为期望输出；a 为实际输出。

此规则的含义是，如果感知机的输出有误，则首先计算误差 e，e 为期望输出与实际输出的差值。新的权值等于旧值加上误差和输入 p 的乘积。偏置则可以看作输入 p 恒为 1 的输入信号所对应的权重，所以新的偏置等于旧偏置加入误差。将偏置理解为一个输入恒为 1 的输入信号的权重是非常有用的，因为这样就从概念上统一了权值和偏置，并使用相

同的算法对它们进行处理。

当计算出新的权重和偏置后，使用测试数据再次测试感知机，直到没有误差或误差在可接受范围内为止。

再让我们回到苹果和香蕉的识别案例上。

设 $w1=1$，$w2=-1$，$b=0$，输入苹果属性 1，1。可以得到：

$$\begin{aligned} net&=p1*w1+p2*w2+b\\ &=1-1+0\\ &=0\\ step(net)&=0 \end{aligned}$$

输出错误，存在误差，所以进行修正。

$$\begin{aligned} e&=t-a\\ &=1-0\\ &=1 \end{aligned}$$

$$\begin{aligned} w_{1new}&=w_{1old}+ep\\ &=1+1*1\\ &=2\\ w_{2new}&=w_{2old}+ep\\ &=-1+1*1\\ &=0\\ b_{new}&=b_{old}+e\\ &=0+1\\ &=1 \end{aligned}$$

使用新的权值带入感知机，重新计算苹果的属性输入。

$$\begin{aligned} net&=p1*w1+p2*w2+b\\ &=1*2+1*0+1\\ &=3\\ step(net)&=1 \end{aligned}$$

纠正误差后，苹果判断正确。尝试判断香蕉。

$$\begin{aligned} net&=p1*w1+p2*w2+b\\ &=-1*2-1*0+1\\ &=-1\\ step(net)&=0 \end{aligned}$$

香蕉判断也正确，误差为 0，学习结束。由此可见，即使初始权值是错误的，但只要

按照感知机学习规则修正权值和偏置，在感知机的可判断范围内，就有可能消除误差或将误差限制在可接受范围内。

感知机的基本原理已经基本介绍完毕。下面可以动手实现自己的感知机了。为了方便读者理解，本书使用 Java 来实现神经网络，并引入神经网络框架 Neuroph 辅助神经网络的实现。有关 Neuroph 框架以及感知机的具体实现，将在下一章中进行详细介绍。

2.3 总结

本章详细介绍了人工神经元的基本模型。作为构成神经网络的基本结构，该模型对学习神经网络有着重要的作用。同时，还介绍了一种最简单的神经网络感知机。虽然最简单的感知机可能只包含单个神经元，但是通过感知机的学习规则，它已经可以进行简单的学习并处理分类问题。

第3章　神经网络框架 Neuroph 介绍

要搭建一个神经网络系统，仅靠理论是不够的，还必须有强大的研发工具支持。在本章中，主要介绍 Neuroph 这一神经网络框架，同时，作为基础，也将简要介绍 Java，Eclipse，Maven 等研发工具。在本章中，你可以知道：

- Neuroph 的基本功能；
- Neuroph Studio 的使用；
- Neuroph 的架构体系；
- 如何使用 Eclipse 进行 Neuroph 开发。

3.1 Neuroph 是什么

Neuroph 是一个轻量级的 Java 神经网络框架，使用它可以很容易地开发基于神经网络的应用，也可以测试各类神经网络。在本书中介绍的所有神经网络的实现，包括 BP 神经网络、Hopfield 网络、SOM 网络等都可以用它实现。

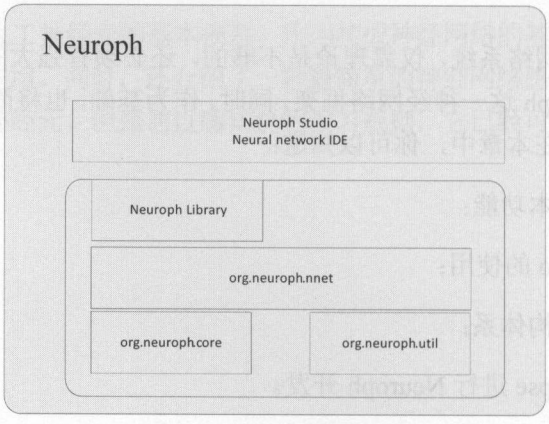
Neuroph 最大特点是对神经网络进行了简单、直观的建模。与其他神经网络框架相比，Neuroph 的模型更容易理解和使用。几乎所有神经网络结构中的概念都可以在 Neuroph 中找到对应的模型，因此初学者特别容易理解。同时，Neuroph 也是一款基于 Apache 2.0 许可的软件，这意味着你可以在任何场景下无偿使用它，无需缴纳任何费用。

Neuroph 是基于 Java 的软件，而 Java 是目前最为流行的大数据与分布式平台之一，这意味着如果有需要，Neuroph 可以与大数据软件无缝集成，运行在分布式平台上。

基于上述原因，本书将重点介绍 Neuroph 框架。

3.2 Neuroph 系统的构成

Neuroph 系统主要由两部分组成，一是 Neuroph Studio，二是 Neuroph Library。Neuroph Studio 是一个用户友好的 GUI 界面，通过它，可以以所见即所得的方式创建、训练和测试神经网络。Neuroph Library 为底层库，Neuroph Studio 依赖于 Neuroph Library。它们的关系如图 3-1 所示。



▲图 3-1　Neuroph 系统架构图

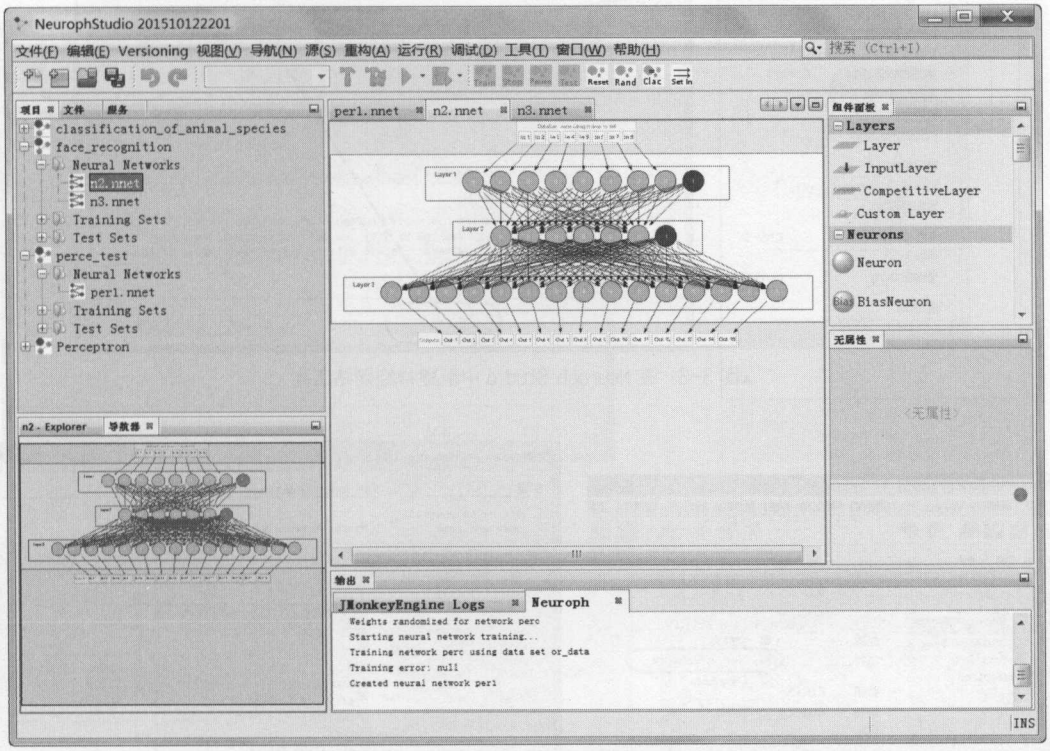
Neuroph Library 是 Neuroph 的核心，它使用 Java 实现了一个基本的神经网络框架，主要包括三个 Java 包。

- org.neuroph.nnet：封装了已经实现好的神经网络，包括感知机、BP 网络、SOM 网络等。
- org.neuroph.core：封装了神经网络的核心模型，如神经元、神经元连接、学习算法抽象模型等。
- org.neuroph.util：封装了一些常用的工具类，如文件读写、神经元工厂等。

3.3 Neuroph Studio 的功能展示

Neuroph Studio 是一个所见即所得神经网络实验工具箱。利用它，你可以很容易地构

造一个自己的神经网络，并用来处理数据。Neuroph Studio 本身是基于 Netbeans 开发的。图 3-2 所示显示了 Neuroph Studio 的工作界面。



▲图 3-2 Neuroph Studio 工作界面

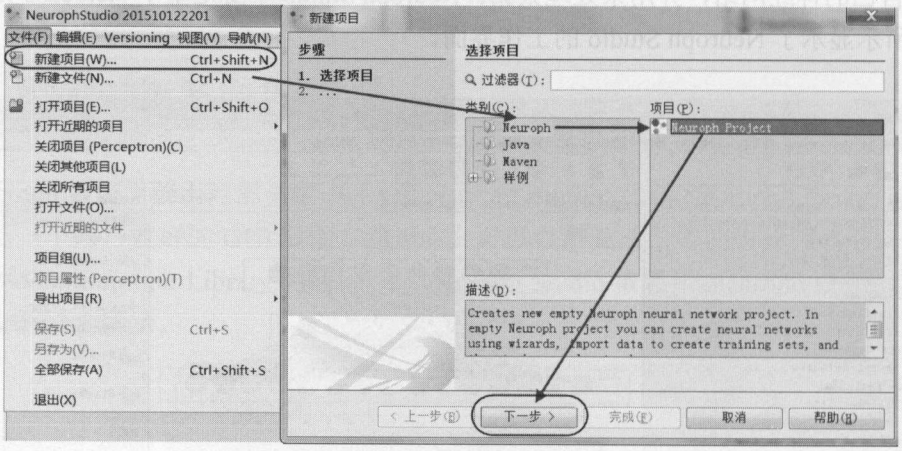
3.3.1 使用 Neuroph Studio 构造感知机处理逻辑与

在上一章中，我们已经可以构造感知机来处理逻辑与问题。同时，也介绍了感知机的学习算法。因此，可以得知，只要我们构造好训练数据，对感知机进行训练，就一定能得到一个合理的感知机使它可以处理逻辑与问题（当然也包括所有和逻辑与问题等价的问题）。使用 Neruoph Studio 可以通过用户界面创建一个感知机网络。

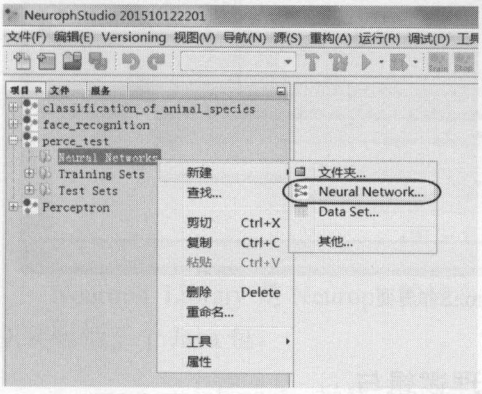
首先，我们需要在 Neuroph Studio 中建立一个神经网络工程，如图 3-3 所示。

接着，在工程的神经网络文件夹下新建神经网络，如图 3-4 所示。

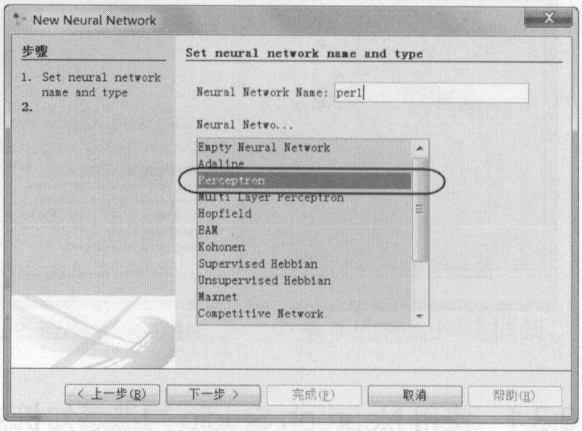
你可以找到多种神经网络的类型，在这里选择感知机（Perceptron），如图 3-5 所示。



▲图 3-3　在 Neuroph Studio 中新建神经网络工程



▲图 3-4　在工程的神经网络文件夹下新建神经网络



▲图 3-5　建立感知机网络

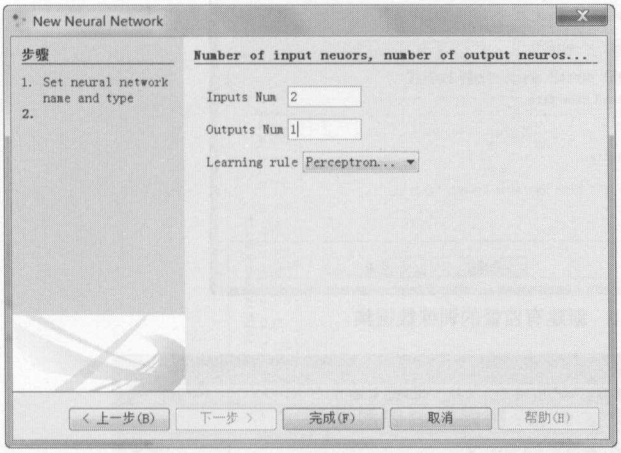
接着，需要输入感知机的输入向量长度和输出向量长度。对于逻辑与操作而言，输入参数显然是两个，输出结果是一个。因此填写 2 和 1，并选择学习算法为感知机学习算法，如图 3-6 所示。

当完成以上操作后，一个感知机就出现在眼前了！如图 3-7 所示。

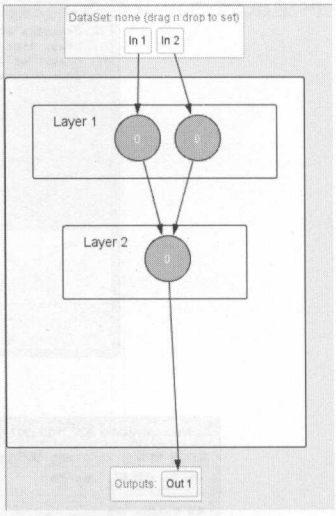
点击输入层和神经元之间的连接，就可以在右边的属性框中看到连接的权值。初始时，权值为随机数。如图 3-8 所示。

为了训练感知机，还需要准备训练数据。在训练集上右键，并新建数据集，如图 3-9

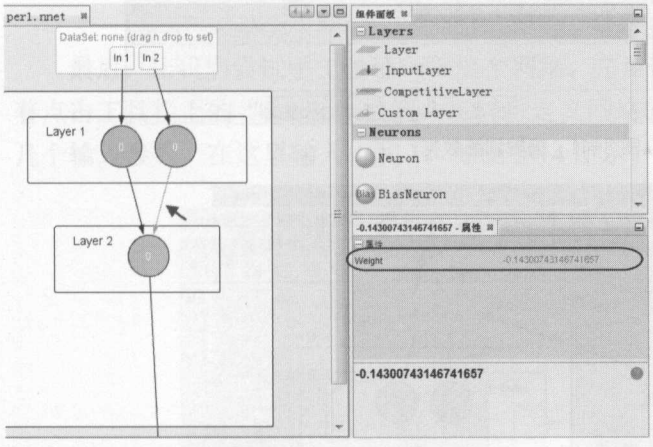
所示。



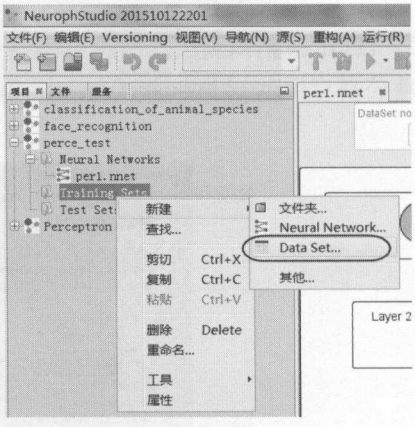
▲图 3-6 设置感知机参数



▲图 3-7 新建立的感知机



▲图 3-8 查看神经元的权重

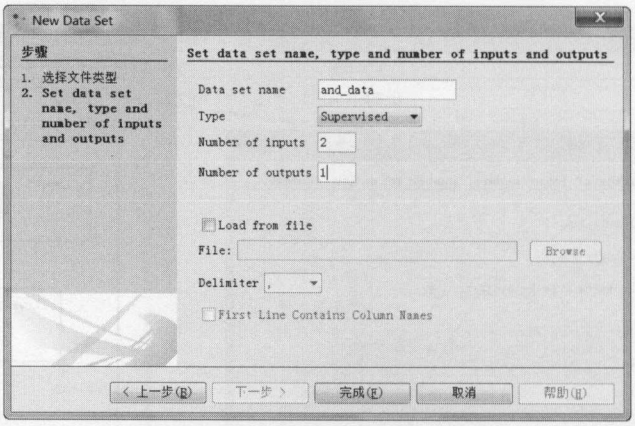


▲图 3-9 新建训练数据集

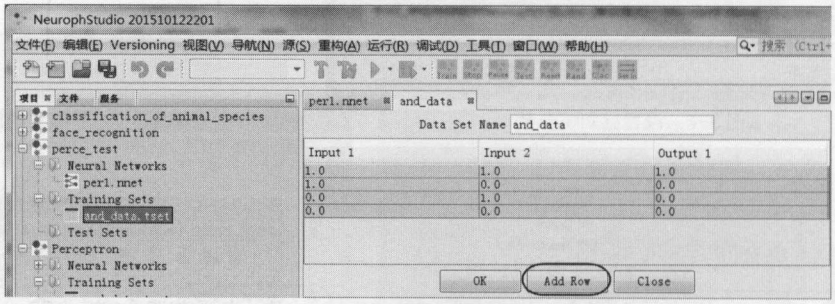
数据类型为有监督的，输入个数为 2，输出为 1，必须与神经网络完全对应，如图 3-10 所示。

通过“Add Row”按钮增加并编辑数据，如图 3-11 所示。

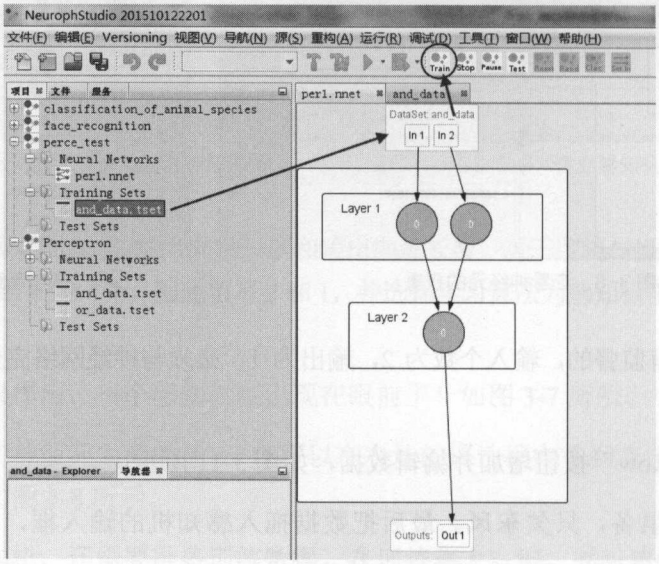
至此，万事俱备，只欠东风。最后把数据拖入感知机的输入框，并点击训练按钮，如图 3-12 所示。



▲图 3-10　新建有监督的训练数据集

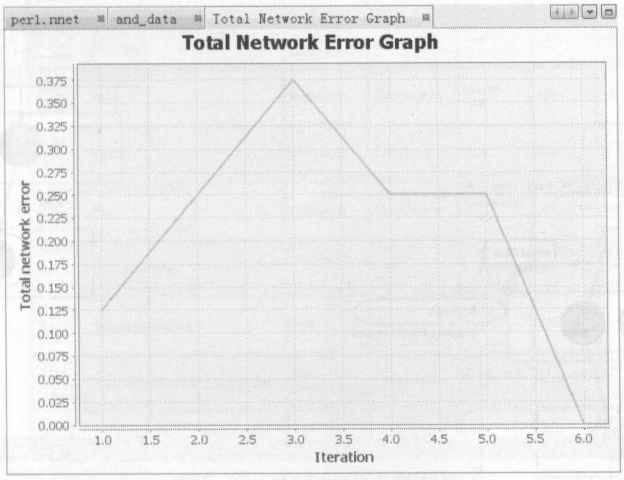


▲图 3-11　编辑训练数据



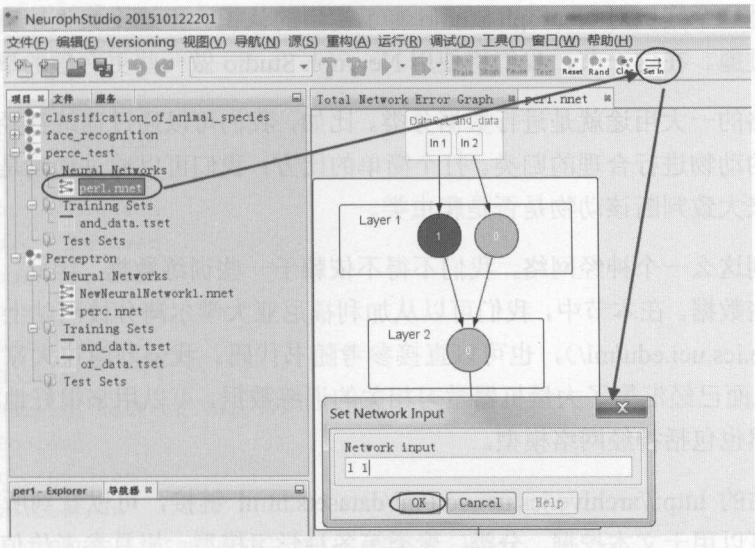
▲图 3-12　使用训练数据训练感知机

训练后，默认情况下会打开训练过程中的误差展示。如图 3-13 所示，可以看到在本次训练中，经过 6 次迭代后，网络的总体误差为 0。这表示，此时该感知机已经完全可以正确记忆逻辑与操作了。



▲图 3-13　训练过程中的误差变化图

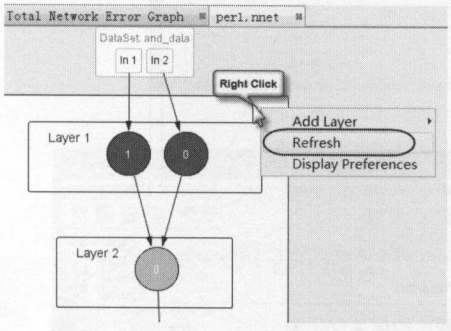
最后，让我们尝试手工验证一下这个网络。先点击左侧导航树上的神经网络节点，接着点击工具栏上的“Set in”按钮，可以自定义网络的输入。输入时，可以使用空格分割几个输入参数。在这里输入 1 和 1。如图 3-14 所示。



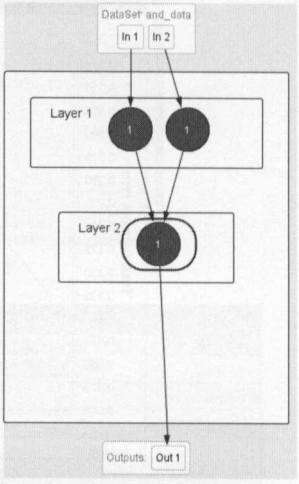
▲图 3-14　自定义网络输入

点击确定按钮后，网络不会立即变化。此时，需要在网络中进行手工刷新。这一步很重要，否则网络不会重新计算新的输入。如图 3-15 所示。

刷新网络后，可以看到，感知机的输出值由 0 变成了 1，如图 3-16 所示。



▲图 3-15　刷新网络



▲图 3-16　刷新网络后的效果

3.3.2　使用 Neuroph Studio 进行动物分类实验

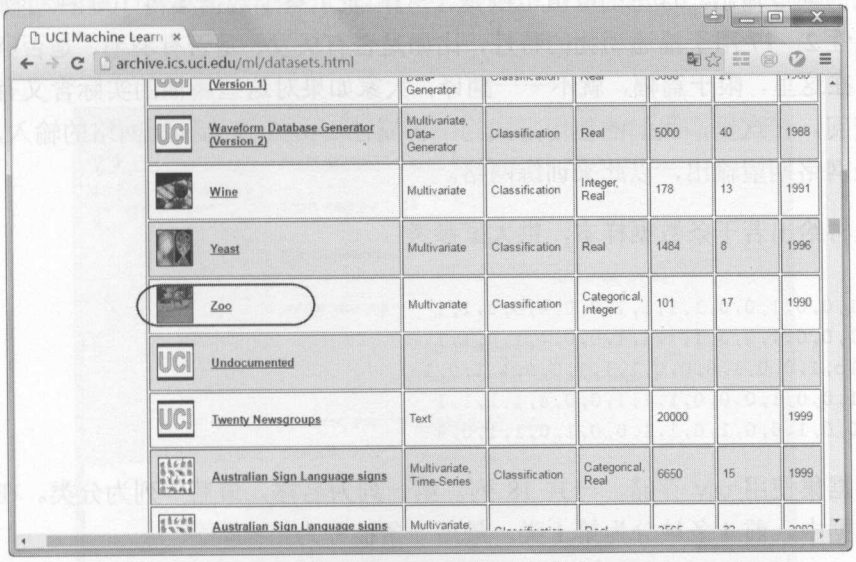
在上一节，我们使用 Neuroph Studio 成功训练了一个可以记忆逻辑与的感知机。相信大家已经通过这个案例对 Neuroph Studio 有了初步的认识。但这个案例毕竟太过于简单，让人有点不过瘾。在这一节，我们将使用 Neuroph Studio 做一些更有趣的事情。

神经网络的一大用途就是进行数据分类。比如，我们可以通过训练神经网络，让网络可以对给定的动物进行合理的归类。打个简单的比方：我们可以通过动物是否是卵生，以及腿的数量来大致判断该动物是否是爬虫类。

为了得到这么一个神经网络，我们不得不依赖于一些训练数据。因此，首先，我们必须先得到这些数据。在本节中，我们可以从加利福尼亚大学尔湾分校网站上获得这些数据（http://archive.ics.uci.edu/ml/），也可以直接参考随书代码。我强烈建议大家可以关注这个网站，因为上面已经汇聚了大量机器学习相关的训练数据，可以用来很好地检验我们的学习模型，当然也包括神经网络模型。

进入网站的 http://archive.ics.uci.edu/ml/datasets.html 链接，可以看到所有的数据集，这些数据集可以用于文本挖掘、分类、聚类等各种学习模型，极具参考价值。在这里，可

以很容易地找到名为 zoo 的数据集。点击进入该数据集，如图 3-17 所示。



▲图 3-17　从 UCI 下载训练数据

点击 zoo，还可以查看对该数据集的描述。从描述可以看到，该数据集给出了大约 100 种动物的属性特点以及它们的分类。给出的属性如下：

```
1. animal name: Unique for each instance
2. hair:   Boolean
3. feathers:   Boolean
4. eggs:   Boolean
5. milk:   Boolean
6. airborne:   Boolean
7. aquatic:Boolean
8. predator:   Boolean
9. toothed:    Boolean
10. backbone:  Boolean
11. breathes:  Boolean
12. venomous:  Boolean
13. fins:Boolean
14. legs:Numeric (set of values: {0,2,4,5,6,8})
15. tail:Boolean
16. domestic:Boolean
17. catsize:Boolean
18. type:  Numeric (integer values in range [1,7])
```

其中，属性 1 表示动物的名字，属性 18 表示动物的分类，也就是该动物的标记(Label)，即神经元的期望输出。这里的取值范围是 1～7，表示整个数据集将 100 种动物分为 7 个类别。属性 2～17 用于描述动物的特性，比如是否有头发、是否有羽毛、是否卵生、有几条腿等。在这里，限于篇幅，就不一一翻译，大家如果对这些属性的实际含义有兴趣，可以自行查阅。在这里，我们的使用方式为：将属性 2～17 作为神经网络的输入，属性 18 作为神经网络期望输出，以此来训练网络。

这里再给出若干条数据样本，供大家参考：

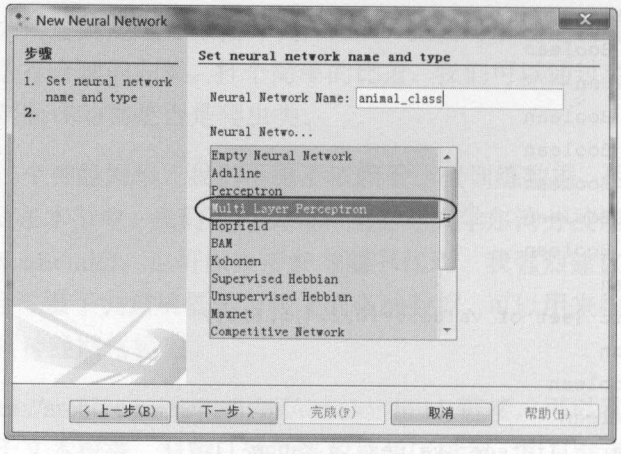
```
bear,1,0,0,1,0,0,1,1,1,1,0,0,4,0,0,1,1
boar,1,0,0,1,0,0,1,1,1,1,0,0,4,1,0,1,1
buffalo,1,0,0,1,0,0,0,1,1,1,0,0,4,1,0,1,1
calf,1,0,0,1,0,0,0,1,1,1,0,0,4,1,1,1,1
carp,0,0,1,0,0,1,0,1,1,0,0,1,0,1,1,0,4
```

该数据集使用 csv 存储，一共 18 列，第一列为名称，最后一列为分类。在给出的 5 条样本数据中，前 4 条被分为第 1 类，最后一条被分为第 4 类。

在进行神经网络训练时，采用 90%的数据进行训练，预留 10%的数据进行网络的验证，用以评估网络的可靠性。

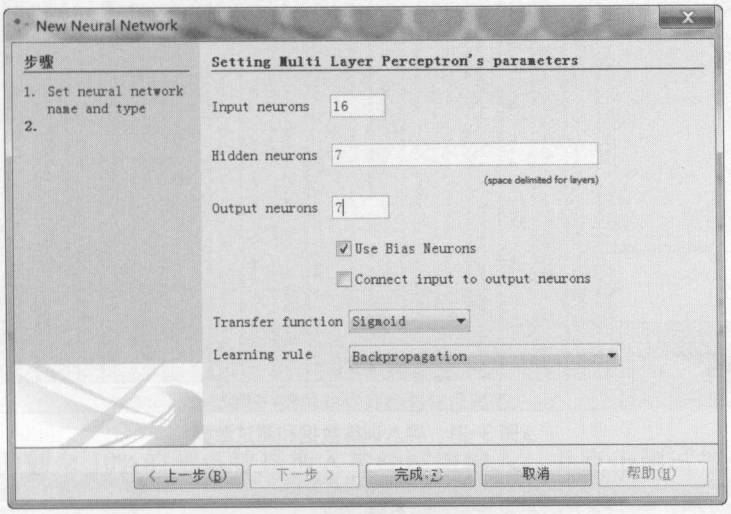
介绍到这里，终于可以步入正题，让我们一起构造这个有趣的神经网络吧！(在这里将构造一个 BP 神经网络。有关 BP 网络的详细描述可以参考本书第 6 章：多层感知机学习算法——BP 学习算法)

首先，新建神经网络，选择多层感知机，如图 3-18 所示。



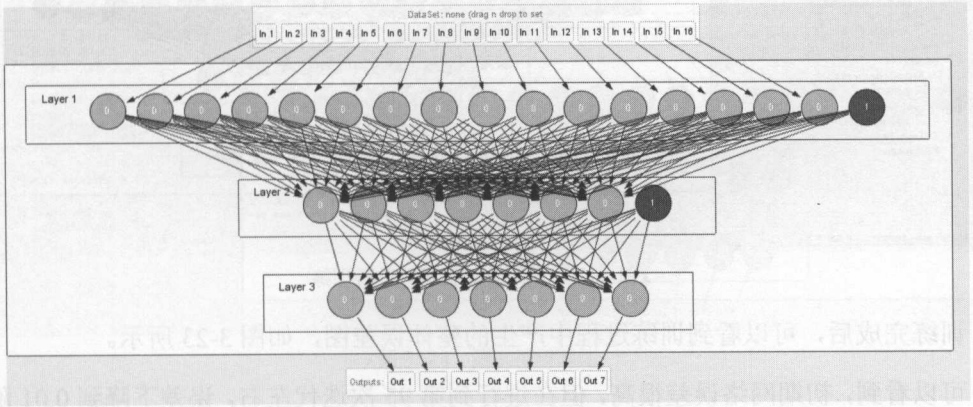
▲图 3-18　选择多层感知机

如上所述，我们有 18−2=16 个输入，因此输入神经元有 16 个，输出神经元有 7 个（每一个类型使用一个神经元表示），隐藏层也使用 7 个神经元（可以调节），如图 3-19 所示。



▲图 3−19　设置神经网络参数

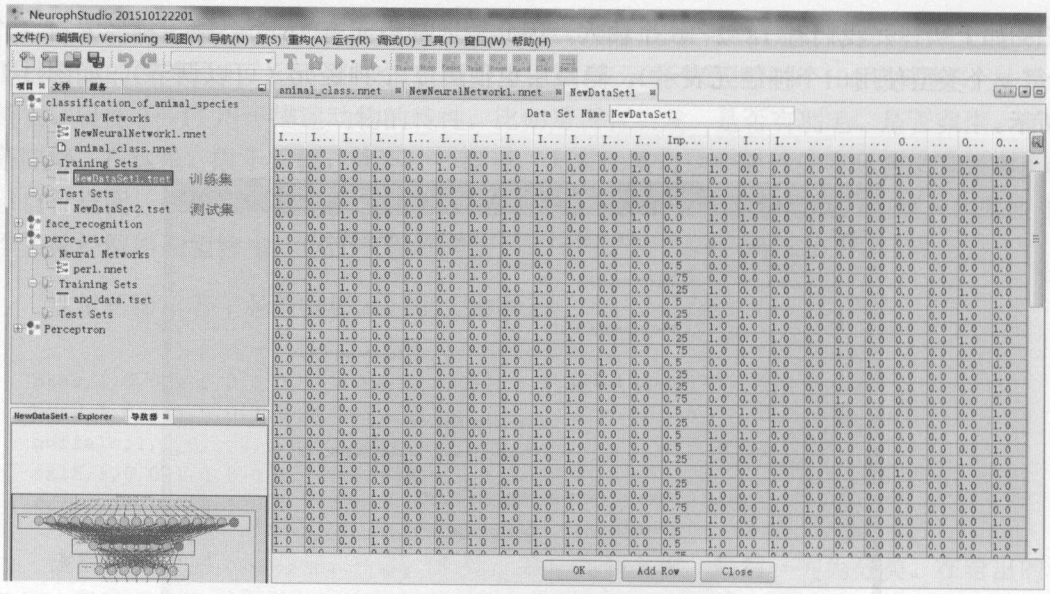
生成的神经网络如图 3-20 所示。



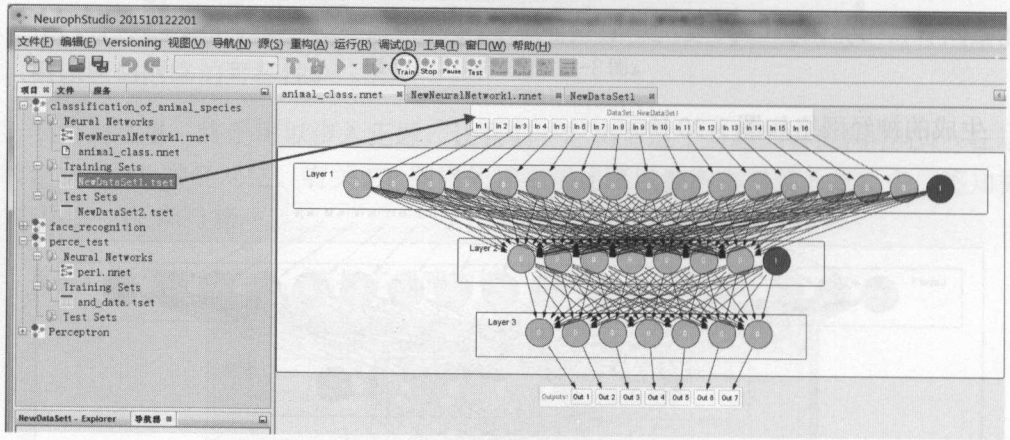
▲图 3−20　BP 神经网络

接着，将我们已经准备好的 90%的数据导入 Neuroph Studio 作为训练集，如图 3-21 所示。

将训练集拖入神经网络并进行训练，如图 3-22 所示。



▲图 3-21　导入训练数据和测试数据

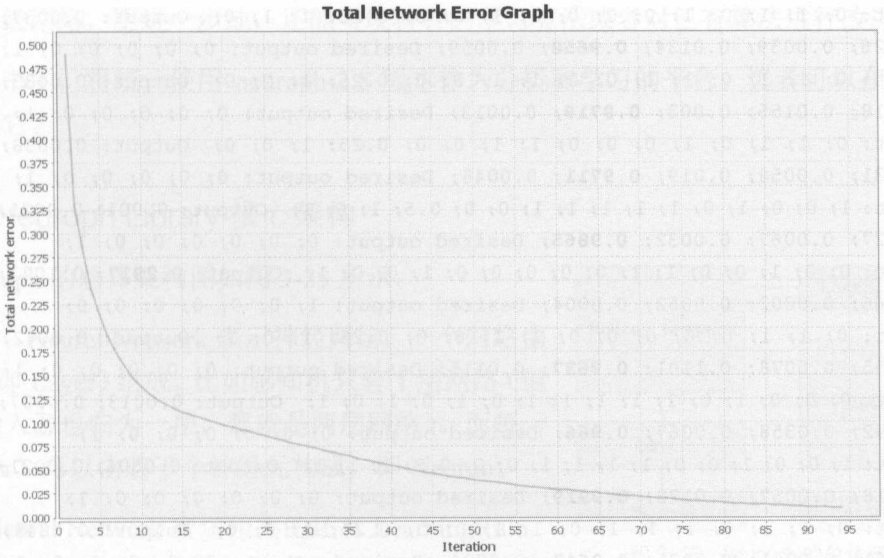


▲图 3-22　关联训练数据并进行训练

训练完成后，可以看到训练过程中产生的整体误差图，如图 3-23 所示。

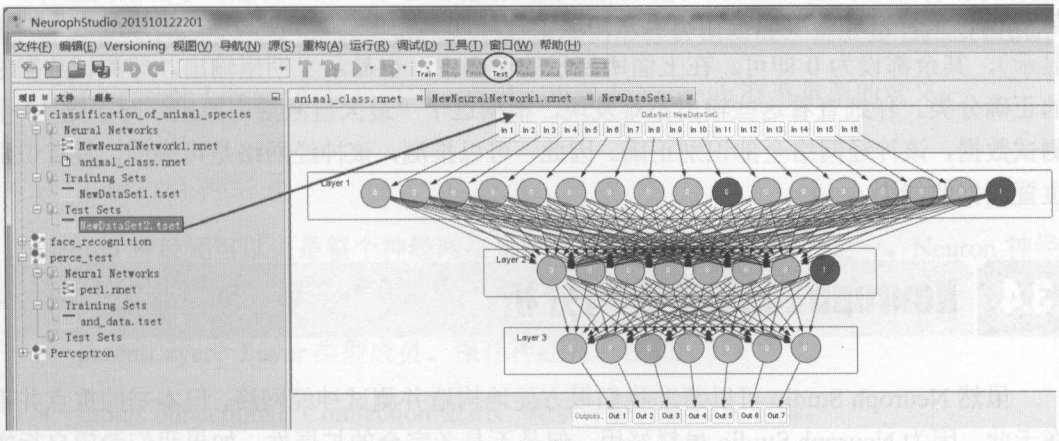
可以看到，初期网络误差很高，但在进行到第 95 次迭代左右，误差下降到 0.01 附近，满足最小误差条件而停止训练（最小期望误差在训练时可以设置，限于篇幅本书没有给出截图，读者自行操作后，不难找到该选项）。

到这里，神经网络已经构造完成。那么它的效果究竟如何呢？如果遇到一个从来没有训练过的动物，这个神经网络可以正确地将它进行归类吗？让我们一试便知！



▲图 3-23 动物分类整体误差图

将准备好的剩余 10%的测试数据拖入神经网络输入，并点击测试按钮，如图 3-24 所示。



▲图 3-24 测试神经网络

测试结果的部分输出如下：

```
Input: 1; 0; 0; 1; 0; 0; 1; 1; 1; 1; 0; 0; 0.5; 1; 0; 1;  Output: 0.0008; 0.0023; 0.0009;
0.0021; 0.0065; 0.0061; 0.9895; Desired output: 0; 0; 0; 0; 0; 0; 1;
Input: 1; 0; 0; 1; 0; 0; 1; 1; 1; 1; 0; 0; 0.5; 1; 0; 1;  Output: 0.0008; 0.0023; 0.0009;
0.0021; 0.0065; 0.0061; 0.9895; Desired output: 0; 0; 0; 0; 0; 0; 1;
```

```
Input: 0; 1; 1; 0; 1; 0; 0; 0; 1; 1; 0; 0; 0.25; 1; 1; 0;  Output: 0.0037; 0.0013;
0.0026; 0.0039; 0.0134; 0.9658; 0.0059; Desired output: 0; 0; 0; 0; 0; 1; 0;
Input: 0; 1; 1; 0; 1; 1; 0; 0; 1; 1; 0; 0; 0.25; 1; 0; 0;  Output: 0.0033; 0.0044;
0.0518; 0.0155; 0.003; 0.9719; 0.0013; Desired output: 0; 0; 0; 0; 0; 1; 0;
Input: 0; 1; 1; 0; 1; 0; 0; 0; 1; 1; 0; 0; 0.25; 1; 0; 0;  Output: 0.0038; 0.0011;
0.0031; 0.0058; 0.019; 0.9711; 0.0048; Desired output: 0; 0; 0; 0; 0; 1; 0;
Input: 1; 0; 0; 1; 0; 1; 1; 1; 1; 1; 0; 0; 0.5; 1; 0; 1;  Output: 0.001; 0.0025; 0.0012;
0.0027; 0.0067; 0.0032; 0.9865; Desired output: 0; 0; 0; 0; 0; 0; 1;
Input: 0; 0; 1; 0; 0; 1; 1; 0; 0; 0; 0; 0; 1; 0; 0; 1;  Output: 0.2917; 0.1071; 0.0509;
0.0066; 0.0002; 0.0052; 0.0004; Desired output: 1; 0; 0; 0; 0; 0; 0;
Input: 0; 1; 1; 0; 0; 0; 0; 0; 1; 1; 0; 0; 0.25; 1; 0; 1;  Output: 0.0042; 0.0003;
0.0055; 0.0078; 0.1101; 0.9637; 0.0115; Desired output: 0; 0; 0; 0; 0; 1; 0;
Input: 0; 0; 0; 1; 0; 1; 1; 1; 1; 1; 0; 1; 0; 1; 0; 1;  Output: 0.0013; 0.0007; 0.0023;
0.0192; 0.0358; 0.0067; 0.966; Desired output: 0; 0; 0; 0; 0; 0; 1;
Input: 1; 0; 0; 1; 0; 0; 1; 1; 1; 1; 0; 0; 0.5; 1; 1; 1;  Output: 0.0006; 0.0021; 0.0008;
0.0018; 0.0057; 0.0122; 0.9919; Desired output: 0; 0; 0; 0; 0; 0; 1;
Input: 0; 1; 1; 0; 1; 1; 1; 0; 1; 1; 0; 0; 0.25; 1; 0; 0;  Output: 0.0048; 0.0037;
0.0161; 0.0061; 0.0031; 0.9647; 0.0024; Desired output: 0; 0; 0; 0; 0; 1; 0;
```

如何解读这些数据呢？由于在输出层使用的 Sigmoid 函数，因此输出结果是连续的，并非像 Step 函数那样为离散的 0 和 1。但由于在输出结果中 7 个输出神经元是完全互斥的，有意义的输出比如是 6 个神经元为 0（未激活）、1 个神经元为 1（激活），因此在解读这些 Output 时，只要简单地将输出最大的那个神经元记为 1（为方便阅读，已将最大值加粗显示），其余都设为 0 即可。在上面所展示的输出中，也显示了期望输出，即测试数据中的正确分类。仔细查看这些输出不难发现，根据这个“最大值激活”原则，对于这 10 个测试数据，该神经网络全部识别正确。因此，可以推测，该神经网络是可靠的，具有识别位置数据的能力。

3.4 Neuroph Library 架构分析

虽然 Neuroph Studio 可以帮助我们很方便地构造并测试神经网络，但本书的重点并不在于此。因为 Neuroph Studio 虽然好用，但是不具备完全的扩展性。如果我们希望自行实现或者修改某个神经网络算法，使用 Neuroph Studio 就会显得力不从心。因此，非常有必要对 Neuroph Studio 进行更深一步的挖掘。在本节中，将介绍 Neuroph Studio 的核心——Neuroph Library。

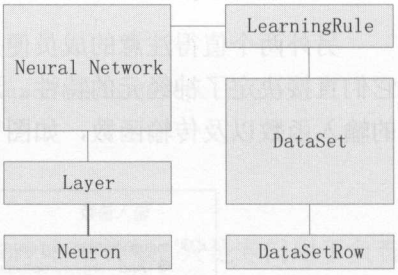
Neuroph Library 是使用 Java 实现的神经网络开发库，Neuroph Studio 的全部功能均有 Neuroph Library 实现。在了解 Neuroph Library 之后，不仅能帮助我们通过编程的方式快

速构造神经网络，更进一步地，还能实现特有的学习算法或者对已有算法进行改进。

在本书中，将统一使用 neuroph-2.8 版本作为分析和学习的平台。读者可以在随书代码中找到它。

3.4.1 Neuroph Library 核心架构

Neuroph 的总体架构图如图 3-25 所示。



▲图 3-25 Neuroph 的总体架构图

其中，Neural Network 表示神经网络，一个网络由若干层（即 Layer）组成，比如感知机只有 1 层网络（但如果把输入源也归为一层，那就是两层网络）。而每一层（Layer）可以由若干个神经元构成，即 Neuron。

与 Neural Network 密切相关的还有 LearningRule，LearningRule 表示该神经网络的学习算法。在 Neuroph 中，不同的网络有不同的 LearningRule 的实现，从面向对象的角度上看，LearningRule 是一个抽象类。

神经网络在学习过程中的一个必要元素就是训练数据。在 Neuroph 中，所有的训练数据都被封装在 DataSet 中，即数据集，一个数据集由多行数据构成，也就是 DataSetRow。

了解 Neuroph Library 的核心架构，对进一步深入 Neuroph 有着重要的意义。

3.4.2 Neuron 神经元

Neuron 神经元可以说是整个神经网络系统最核心、最基础的组件之一。Neuron 神经元内部维护这神经元所必需的所有信息，其核心内部成员如下。

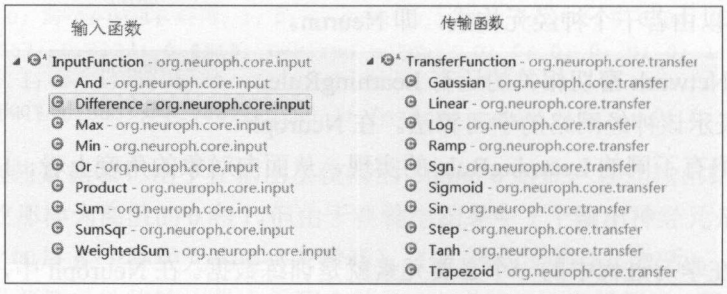
- parentLayer：Layer 类型成员。保存神经元所在的层。
- inputConnections：Connection 数组，表示神经元的输入连接。
- outConnections：Connection 数组，表示神经元的输出连接。
- netInput：double 类型，表示神经元的净输入。
- output：double 类型，表示神经元的净输出。
- error：double 类型，保存神经元的误差。

- inputFunction：InputFunction 类型，表示神经元的输入函数，通常为加权求和。
- transferFunction：TransferFunction 类型，表示神经元的传输函数。

其中 Connection 类型封装了神经元连接，对于每一个 Connection 类型的对象，主要包含源神经元 fromNeuron、目的神经元 toNeuron 和连接权重 weight 三部分信息。

此外，netInput 表示净输入，即输入函数 inputFunction 的输出结果。

另外两个值得注意的成员便是输入函数 InputFunction 和传输函数 TransferFunction，它们直接决定了神经元的特性。通过 Eclipse 的辅助，可以很容易找到在 Neuroph 中支持的输入函数以及传输函数，如图 3-26 所示。



▲图 3-26　Neuroph 支持的输入函数和传输函数

在后续的章节中将根据各个场景逐一介绍这些函数的使用，在这里仅简单了解即可。

为了创建一个神经元，Neuroph 提供了一个工厂类 NeuronFactory，可以使用它创建给定条件的神经元，比如如下示例代码：

```
NeuronProperties neuronProperties = new NeuronProperties();
neuronProperties.setProperty("transferFunction", TransferFunctionType.SIGMOID);
Neuron neuron = NeuronFactory.createNeuron(neuronProperties);
```

可以看到，使用 NeuronFactory 创建神经元需要给定一个 NeuronProperties 对象。该对象用于设置神经元的各项参数，比如指定传输函数、输入函数，也可以指定传输函数的参数等。

3.4.3　Layer 层

Layer 代表神经网络中的一层，也就是多个神经元的一个有序集合。在 Layer 中，主要保存神经网络 NeuralNetwork 以及该层包含的神经元集合两个信息。其中，

NeuralNetwork 为当前层所在的网络。

Layer 提供了一些操作用于增加或者删除神经元，比如，下面的 Layer 构造函数，可以看到，在使用 NeuronFactory 创建神经元后，调用了 addNeuron()方法将神经元加入层中：

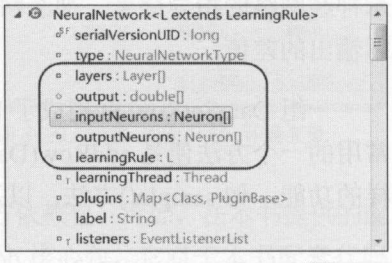
```
public Layer(int neuronsCount, NeuronProperties neuronProperties) {
    this();
    for (int i = 0; i < neuronsCount; i++) {
            Neuron neuron = NeuronFactory.createNeuron(neuronProperties);
            this.addNeuron(neuron);
    }
}
```

创建 Layer 可以使用 LayerFactory。它与 NeuronFactory 非常类似，只是专门用于负责创建 Layer。比如，下面的示例代码使用给定的 neuronProperties 属性，创建了一个包含两个神经元的层：

```
Layer outputLayer = LayerFactory.createLayer(2, neuronProperties);
```

3.4.4 NeuralNetwork 神经网络

NeuralNetwork 表示神经网络本身，主要包含层信息（Layer 数组）、网络输出（output），输入神经元组（inputNeurons）、输出神经元（outputNeurons）以及学习算法（learningRule）。与 Layer 类似，它还提供了一些方法用于向网络中增加或者删除层。

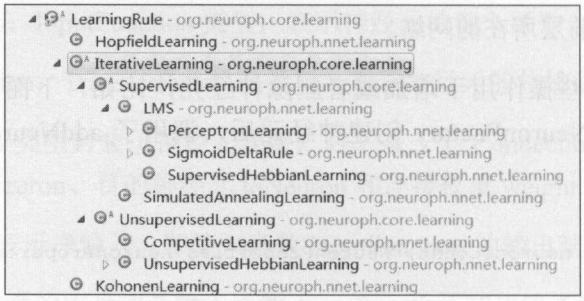


▲图 3-27 NeuralNetwork 结构

除了维护这些静态结构外，NeuralNetwork 中维护的学习算法 LearningRule 也至关重要。它直接决定了神经网络的学习方式。图 3-27 所示显示了 NeuralNetwork 的所有静态数据，其中矩形框内为其核心数据结构。

3.4.5 LearningRule 学习算法

如果说 NeuralNetwork 是神经网络的身体，那么 LearningRule 就是整个神经网络的大脑，它告诉神经网络应该如何学习和计算。Neuroph 中，已经内置大部分学习算法，比如 Hopfield 算法、Kohonen 算法、感知机学习算法、BP 算法等，如图 3-28 所示。



▲图 3-28　LearningRule 的主要学习算法

从类的层次上看，迭代学习算法（Iterative Learning）的子类最为丰富，因为大部分的学习算法需要迭代反复执行。IterativeLearning 有两个重要的子类，即有监督的学习（SupervisedLearning）和无监督的学习（UnsupervisedLearning）。有监督的学习，表示训练数据中有指导信息，即打标数据，也就是说训练数据已经给出某一条数据的正确分类，这些信息可以更好地指导神经网络学习；在无监督的学习中，训练数据没有给出指导信息，网络无法通过反馈计算误差来指导学习过程。在本书中，介绍的感知机学习算法、BP 算法等属于有监督的学习，Hopfield 算法、Kohonen 算法、Sanger 算法等属于无监督的学习。

3.4.6　DataSet 和 DataSetRow

DataSet 和 DataSetRow 用于保存训练数据。DataSet 表示数据集，DataSetRow 为数据集中某一行。DataSetRow 中最重要的为数据输入 input 和期望输出 desiredOutput 两个成分，它们都是 double 数组。对于有监督的学习而言，desiredOutput 通常是必须设置的，它表示神经网络的指导信号。神经网络的神经元误差通常就是指期望输出 desiredOutput 和实际输出的差值。

一组 DataSetRow 就构成了 DataSet。DataSet 可以用于管理 DataSetRow。最重要也最常用的一个方法便是 addRow(DataSetRow row)。此外，DataSet 还提供了对数据集进行采样的功能，如 sample()方法，以及数据归一化的功能，如 normalize()方法。

3.5　Neuroph 开发环境搭建

到目前为止，大家应该对 Neuroph 的总体代码结构有了一定的了解。是不是有点跃跃欲试了呢？在本节中，将详细介绍 Neuroph 开发环境的搭建。对于已经有 Java 开发经验的读者，大可以跳过本节，只需要找到随书代码即可，里面已经包含了 Neuroph 库以及本

书所有可运行示例。将源码工程简单地导入 Eclipse 或者 IDEA 等 IDE 环境，即可进行开发工作。

对于不熟悉 Java 开发的读者，可以详读本节，搭建属于自己的神经网络开发环境。

3.5.1 基础平台——Java 介绍以及安装

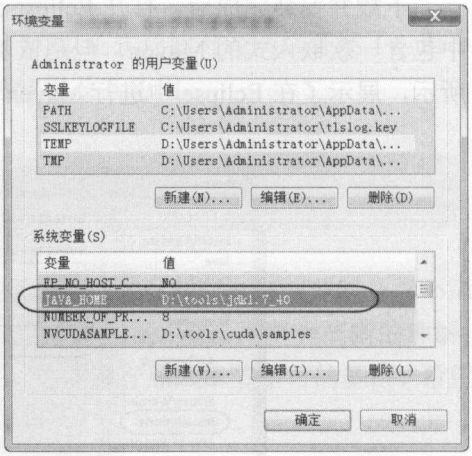
Java 是目前最为流行的计算机语言之一，大量应用于企业级软件开发。目前最为流行的大数据平台 Hadoop 正是使用 Java 构建的。因此，使用 Java 开发神经网络系统天生就可以与这些数据平台进行良好的交互。同时，Java 也是一门易于学习，并适合大规模、多人多团队协作开发的一门语言，这些也正是 Neuroph 选择 Java 的巨大优势。

读者可以在 Oracle 的官方网站 http://www.oracle.com/technetwork/java/javase/downloads/index.html 上下载 Java 的开发包 JDK（Java Development Kit）。

在本书创作过程中，最新的版本为 JDK 8。本书使用的版本为 JDK 7。这两个版本都可以用于运行本书的示例代码。

下载 JDK 的安装文件后，双击安装即可，过程从略。在安装完成后，建议设置 JAVA_HOME 环境变量指向 JDK 的安装目录，如图 3-29 所示。

最后，还需要将“%JAVA_HOME%/bin”目录加入系统的 PATH 变量中。



▲图 3-29 设置 JAVA_HOME 环境变量

3.5.2 包管理工具——Maven 安装

Maven 是 Java 的一款构建以及依赖管理工具。在使用 Java 进行软件开发的过程中，必然会涉及不少第三方依赖包。比如，在本书提供的配套代码中，程序就必须依赖 neuroph-core-2.8.jar 这个 Neuroph 核心包，它对于本书配套代码来说，就属于第三方依赖。此外，程序还会依赖如 slf4j 等一些基础日志包，使用 Maven 就可以很方便地管理所有的依赖。因此，在这里选择 Maven 工程作为本书配套代码的组织形式。

如果读者对 Maven 完全没有概念，也不必惊慌，因为本书代码已经建立并且组织好了一个完整的 Maven 工程，读者完全不必费心在这方面自己动手，只需要简单地导入

Eclipse 即可，依然可以把主要的精力放置在代码的学习和调试上。

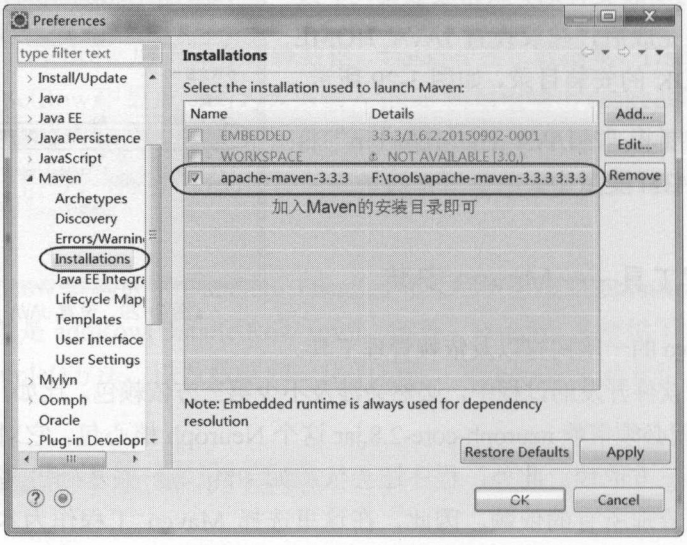
Maven 作为一款开源免费的软件，可以很容易地在 Apache 官方网站 https://maven.apache.org/上获得。

读者可以下载最新的 Maven 版本，无需安装，解压即可。

3.5.3　开发工具——Eclipse 安装

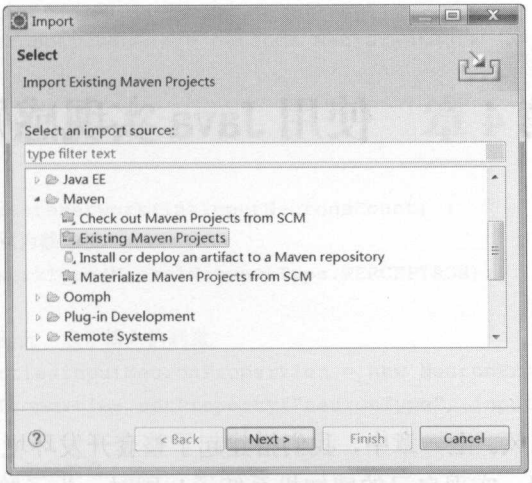
Eclipse 是使用最为广泛的 Java IDE 环境之一，它是完全免费的。本书的配套代码使用了 Eclipse 作为开发环境。读者可以在 Eclipse 官方网站下载并安装。Eclipse 的官方下载地址为 https://www.eclipse.org/downloads/。强烈推荐大家下载 Java EE 版本，因为这个版本已经预制安装了不少实用的插件。本书所使用的版本为 Eclipse Mars(4.5.0)Java EE IDE for Web Developers。

下载并安装完成后，打开 Eclipse 进行 Maven 配置。虽然 Eclipse 内置的 Maven 插件中包含一款嵌入式的 Maven，但是依然建议大家使用独立的 Maven 进行开发。如图 3-30 所示，展示了在 Eclipse 中进行 Maven 配置。



▲图 3-30　在 Eclipse 中配置 Maven

通过 Windows→Perference 打开首选项对话框。接着就可以导入本书的配套代码。选择 File→Import，然后选择导入已经存在的 Maven 工程，如图 3-31 所示。



▲图 3-31 导入已存在 Maven 工程

导入过程中，选择包含有 pom.xml 文件的目录即可。

导入成功后，开发环境即配置完成。至此，我们可以正式开始神经网络开发了！迫不及待的你就赶快进入下一章节吧！使用 Java 实现感知机及其应用。

3.6 总结

经过本章的学习，读者应该对 Neuroph 框架有了概要性的认识，也许细节部分尚不清晰，但在后续章节中会逐步补充其中缺失的细节，从而对 Neuroph 有一个更为清晰的理解。本章的另一个重要内容是 Neuroph 开发环境的搭建，这是进行后续章节学习的基础，希望读者可以根据书中所述完成开发环境的搭建工作。

第4章 使用Java实现感知机及其应用

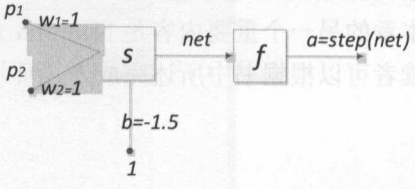
万事俱备，只欠东风。上一章中，我们搭建起了整套开发环境，在本章中，我们可以结合感知机的基本原理，实现自己的感知机系统了！同时，为了验证实现的感知器系统，本章还准备了几个实际案例进行探讨。在本章中，你将学到：

- 如何使用Neuroph创建任意结构的神经网络；
- 如何实现一套学习算法让神经网络拥有智能；
- 感知机的局限性。

4.1 第一个Neuroph程序——使用感知机记忆逻辑与

相信大家一定还记得第2章中提出的一个可以处理逻辑与问题的感知机。为了阅读方便，这里再次给出该感知机的结构图，如图4-1所示。

本节将详细介绍如何通过编码的方式构造这样一个感知机。



▲图4-1 可以记忆逻辑与操作的感知机

4.1.1 创建感知机网络

给定感知机拥有一个神经元、两个输入，并且还有一个神经元偏置，传输函数为*step*函数。基于NeuralNetwork类，构造一个神经网络，并命名为AndPerceptronNoLearn（如果读者已经将工程导入Eclipse，可以使用快捷键Ctrl+Shift+T快速查找到该类的代码）：

```
public class AndPerceptronNoLearn extends NeuralNetwork {
…
```

接着看一下如何创建上述网络：

```
01 private void createNetwork(intinputNeuronsCount) {
02     // 设置网络类别为感知机
03     this.setNetworkType(NeuralNetworkType.PERCEPTRON);
04
05     // 建立输入神经元，表示输入的刺激
06     NeuronPropertiesinputNeuronProperties = new NeuronProperties();
07     inputNeuronProperties.setProperty("neuronType", InputNeuron.class);
08
09     // 由输入神经元构成的输入层
10     Layer inputLayer = LayerFactory.createLayer(inputNeuronsCount, inputNeuronProperties);
11     this.addLayer(inputLayer);
12     // 在输入层增加 BiasNeuron，表示神经元偏置
13     inputLayer.addNeuron(new BiasNeuron());
14
15     //设置传输函数为 step()函数
16     NeuronPropertiesoutputNeuronProperties = new NeuronProperties();
17     outputNeuronProperties.setProperty("transferFunction", TransferFunctionType.STEP);
18     Layer outputLayer = LayerFactory.createLayer(1, outputNeuronProperties);
19     this.addLayer(outputLayer);
20
21     //将输入层和输出层进行全连接
22     ConnectionFactory.fullConnect(inputLayer, outputLayer);
23     NeuralNetworkFactory.setDefaultIO(this);
24     Neuron n = outputLayer.getNeuronAt(0);
25
26     //设置输入神经元和感知机之间的连接权重
27     n.getInputConnections()[0].getWeight().setValue(1);
28     n.getInputConnections()[1].getWeight().setValue(1);
29     n.getInputConnections()[2].getWeight().setValue(-1.5);
30 }
```

上述代码定义了 createNetwork()函数用于创建给定的感知机。第 3 行，设置网络类型为感知机。第 6～10 行定义了输入层，输入层中的神经元为输入神经元，严格意义上来说并不属于整个神经元的组成，它只表示神经网络的输入部分。但考虑建模方便，将输入节点也定义为一种特殊的神经元，即输入神经元 InputNeuron。

第 11 行将由输入神经元组成的输入层加入到当前网络，同时在第 13 行将一个贝叶斯

神经元 BiasNeuron 加入输入层。贝叶斯神经元 BiasNeuron 是一种特殊的神经元，正如第 2 章中提到的，神经元偏置可以理解为输入恒为 1 的输入信号对应的连接权重，这里的 BiasNeuron 正是使用这种思想。在网络中加入 BiasNeuron 神经元，等同于对网络下一层的每个神经元都加上偏置信号。至此，网络的输入层构造完成。

第 16～19 行设置了神经元的传输函数为 *step* 函数。第 22～24 行将输入节点和神经元进行全连接，也就是每个输入节点和每个神经元都进行两两连接。

最后，在第 27～29 行设置每个连接的权重，1 和 1 分别是输入节点到神经元的权值，−1.5 为神经元的偏置。

在神经网络建立之后，就可以测试该网络了。因为权值都是事先设计好的，不存在学习的过程，因此只要简单地给出测试数据即可：

```
01 public static void main(String[] args) {
02   DataSettrainingSet = new DataSet(2, 1);
03   trainingSet.addRow(new DataSetRow(new double[] { 0, 0 }, new double[] { Double.NaN }));
04   trainingSet.addRow(new DataSetRow(new double[] { 0, 1 }, new double[] { Double.NaN }));
05   trainingSet.addRow(new DataSetRow(new double[] { 1, 0 }, new double[] { Double.NaN }));
06   trainingSet.addRow(new DataSetRow(new double[] { 1, 1 }, new double[] { Double.NaN }));
07
08   AndPerceptronNoLearn perceptron = new AndPerceptronNoLearn(2);
09
10   for (DataSetRowrow :trainingSet.getRows()) {
11         perceptron.setInput(row.getInput());
12         perceptron.calculate();
13         double[] networkOutput = perceptron.getOutput();
14         System.out.println(Arrays.toString(row.getInput()) + "=" + Arrays.toString
           (networkOutput));
15    }
16 }
```

这是一段主函数，用于测试刚刚构造的感知机网络。第 2～6 行创建学习数据集，数据集有 2 个输入、1 个输出。由于该网络不需要进行学习，因此数据集中的期望值为 NaN。

第 10～15 行对于每一条测试数据进行计算。第 11 行设置网络的输出，第 12 行执行网络计算，第 13 行取得计算后的输出。最终，这段代码的执行结果如下：

```
[0.0, 0.0]=[0.0]
[0.0, 1.0]=[0.0]
```

```
[1.0, 0.0]=[0.0]
[1.0, 1.0]=[1.0]
```

可以看到，它确实已经正确地识别了所有的逻辑与操作。

4.1.2 理解输入神经元 InputNeuron

输入神经元 InputNeuron 是一种特殊的神经元。严格意义上来说，它并不是神经网络的一部分，而只是表示输入的数据点。由于神经网络在读入数据时，就会有连接权重，因此出于建模的需要，就有了 InputNeuron 这么一类神经元。

它的核心代码非常简单：

```
public class InputNeuron extends Neuron {
    public InputNeuron() {
        super(new WeightedSum(), new Linear());
    }
    @Override
    public void calculate() {
        this.output = this.netInput;
    }
}
```

首先 InputNeuron 是 Neuron 的一种，因此它继承自 Neuron。其次，由于 InputNeuron 必须忠实地反映输入数据，因此它重写了 calculate()方法，直接将净输入作为函数的输出，而净输入本身则通常可以由网络的 setInput()方法直接设置。

4.1.3 理解贝叶斯神经元 BiasNeuron

在前文中已经介绍过，为了统一偏置和神经元连接的权值，可以将偏置视为一个输入恒为 1 的神经元。贝叶斯神经元就是基于这种思想而实现的一个特殊的神经元，其核心代码如下：

```
public class BiasNeuron extends Neuron {
    public BiasNeuron() {
            super();
    }

        @Override
        public double getOutput() {
```

```
            return 1;
        }
        .....
```

可以看到，贝叶斯神经元重写了 getOutput()方法，并返回 1，因为偏置值恒为 1。在学习过程中，系统只是不断地调整贝叶斯神经元和下层神经元之间的连接权重，这也相当于调整了下层神经元的偏置值。如果不需要偏置值，那么也就不需要使用贝叶斯神经元。

4.1.4　*step* 传输函数是如何实现的

传输函数 *step* 可以说是使用得最为广泛的传输函数之一。在 Neuroph 中，所有的传输函数都从抽象类 TransferFunction 中继承而来。传输函数的核心是根据一个净输入来计算神经元的输出。该功能使用 getOutput()方法来实现。

其核心实现如下：

```
01 public class Step extends TransferFunction implements Serializable {
02   private double yHigh = 1d;
03   private double yLow = 0d;
04   public Step() {
05   }
06   public Step(Properties properties) {
07       try {
08             this.yHigh = (Double)properties.getProperty("transferFunction.yHigh");
09             this.yLow = (Double)properties.getProperty("transferFunction.yLow");
10       } catch (NullPointerException e) {
11             // if properties are not set just leave default values
12       } catch (NumberFormatException e) {
13             System.err.println("Invalid transfer function properties! Using
               default values.");
14       }
15   }
16
17      @Override
18    public double getOutput(double net) {
19         if (net > 0d)
20               return yHigh;
21         else
22               return yLow;
23    }
24    ....
```

可以看到，第19～22行为了保证函数的通用性，*step* 函数并没有简单地返回0或者1。而是使用了yHigh和yLow两个变量保存返回的上限值和下限值（默认为1和0）。根据使用的场景可以随意设置这两个值。第 8～9 行，通过构造函数传入 transferFunction.yHigh 和transferFunction.yLow。这种通过property属性进行对象设置的方法，在Neuroph中是非常常见和通用的。

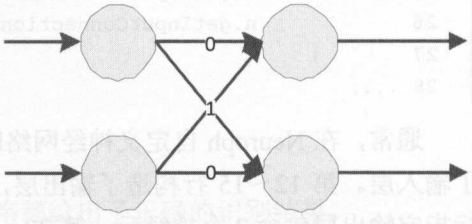
4.2 让感知机理解坐标系统

坐标系统是数学中非常重要的工具。一个平面直角坐标系将空间划分为四块，除了坐标轴上的点，其他点均落在第一、二、三、四象限。在坐标轴之外，给定一个坐标，它必然位于这四个象限中的一个。当然，我们可以很容易地写一个程序，判断一个坐标点位于哪个象限。但在本节中，将使用神经网络的方法，“教会”神经网络“理解”坐标系统，并让它可以正确地判断任何一个坐标所处的位置。

4.2.1 感知机网络的设计

很显然，由于每个坐标点必然归属于一个象限，因此每一个象限可以被视为一个分类。所以，这个问题其实是一个分类问题。但和逻辑与问题不同，在处理逻辑与时，由于结果只可能是0和1，因此是一个简单的二分类问题。但在这里，由于存在4个象限，因此是一个略微复杂的四分类问题。使用单个神经元显然无法识别，因此必须使用两个神经元组成的感知机网络。对于两个神经元输出而言，存在[1 1]，[0 1]，[0 0]，[1 0]四种情况，正好对应第一、二、三、四共4个象限。

在第 2 章已经介绍过如何判定边界与权值向量垂直。这里的判定边界显然是坐标轴本身，因此对应的权值向量可以取[0 1]和[1 0]（任何一个和坐标轴垂直的向量都是可以的）。



▲图 4-2 识别坐标点的象限的感知机

综上所述，可以得到图 4-2 所示的感知机网络。

4.2.2 感知机网络的实现

根据上述网络结构，使用Neuroph可以很容易地实现它。完整的代码可以参考本书源

码中 geym.nn.perceptron.PerceptronClassifyNoLearn。这里给出核心实现的说明：

```
01 public class PerceptronClassifyNoLearn extends NeuralNetwork {
02    private void createNetwork(intinputNeuronsCount) {
03           // 设置网络类别为感知机
04           this.setNetworkType(NeuralNetworkType.PERCEPTRON);
05           // 输入神经元建立，表示输入的刺激
06           NeuronPropertiesinputNeuronProperties = new NeuronProperties();
07           inputNeuronProperties.setProperty("neuronType", InputNeuron.class);
08           // 由输入神经元构成的输入层
09       Layer inputLayer = LayerFactory.createLayer(inputNeuronsCount, inputNeuron
         Properties);
10           this.addLayer(inputLayer);
11
12           NeuronPropertiesoutputNeuronProperties = new NeuronProperties();
13          outputNeuronProperties.setProperty("transferFunction", TransferFunction
            Type.STEP);
14           Layer outputLayer = LayerFactory.createLayer(2, outputNeuronProperties);
15           this.addLayer(outputLayer);
16
17           ConnectionFactory.fullConnect(inputLayer, outputLayer);
18           NeuralNetworkFactory.setDefaultIO(this);
19
20           Neuron n=outputLayer.getNeuronAt(0);
21           n.getInputConnections()[0].getWeight().setValue(1);
22           n.getInputConnections()[1].getWeight().setValue(0);
23
24           n=outputLayer.getNeuronAt(1);
25           n.getInputConnections()[0].getWeight().setValue(0);
26           n.getInputConnections()[1].getWeight().setValue(1);
27       }
28 ....
```

通常，在 Neuroph 自定义神经网络时，需要继承 NeuralNetwork 类。第 6～10 行构造了输入层。第 12～15 行构造了输出层，其中，第 13 行指定输出层传输函数 step，第 14 行指定输出层包含 2 个神经元。第 20～26 行设置了连接权重为[0 1]和[1 0]，这一步尤为重要，网络能否正常工作，直接和权重相关。

接着，就可以使用这个感知机工作了：

```
01 public static void main(String[] args) {
02   Scanner scanner = new Scanner(System.in);
03   String line = null;
```

```
04    double[] input = new double[2];
05    PerceptronClassifyNoLearn perceptron = new PerceptronClassifyNoLearn(2);
06    try {
07        while ((line = scanner.nextLine()) != null) {
08              String[] numbers = line.split("[\\s|,|;]");
09              input[0] = Double.parseDouble(numbers[0]);
10              input[1] = Double.parseDouble(numbers[1]);
11
12              perceptron.setInput(input);
13              perceptron.calculate();
14              double[] networkOutput = perceptron.getOutput();
15              System.out.println(Arrays.toString(input) + "=" + posToString(networkOutput));
16
17          }
18    } finally {
19          scanner.close();
20    }
21 }
```

该感知机接受一个坐标点的输入，然后进行识别，并给出坐标点所在的位置。程序第2～10行读入控制台输入，并解析为神经网络可以识别的格式。第12～13行进行识别。第15行获得识别结果并给出输出，其中posToString()函数将网络的识别结果转为人可读的文字。程序执行后，一种可能的输出如下：

```
1 1
[1.0, 1.0]=第一象限
1 -1
[1.0, -1.0]=第四象限
-1 1
[-1.0, 1.0]=第二象限
-1 -1
[-1.0, -1.0]=第三象限
0.01 -100
[0.01, -100.0]=第四象限
```

可以看到，对于本例中给出的5个输入，网络都给出了正确的识别结果。

4.3 感知机学习算法与Java实现

到此为止，我们的感知机似乎已经可以“做”些什么了。但依然存在一个严重的问题，那就是感知机是被事先设计好的，并不具备任何学习能力。在第2章，我们已经详细介绍

了感知机的学习算法：

$$w_{new}=w_{old}+ep$$
$$b_{new}=b_{old}+e$$

那么，是否可以将这个算法使用 Neuroph 直接实现，从而避免对神经网络的事先设计呢？答案是肯定的。

4.3.1　感知机学习规则的实现

根据 Neuroph 框架的设计思想，所有的算法或学习规则都应该从 LearningRule 类继承。感知机学习算法也不例外。同时，感知机学习算法是典型的迭代监督式学习，而这类学习算法在 SupervisedLearning 类中已经进行了封装。因此，要实现感知机学习规则，可以直接从 SupervisedLearning 继承。

在 SupervisedLearning 中有一个抽象函数 updateNetworkWeights()，它根据每个神经元的误差来调整神经元的权重：

```
abstract protected void updateNetworkWeights(double[] outputError);
```

这里的 outputError 表示神经元的误差，它是训练数据的期望输出与神经元实际输出的差值。由感知机学习规则可知，权重的调整值是误差与神经元输入的乘积。

定义感知机学习规则 PerceptronLearningRule，并结合上述抽象函数以及感知机学习公式，不难得出以下代码：

```
01 protected void updateNetworkWeights(double[] outputError) {
02       int i = 0;
03       for (Neuron neuron :neuralNetwork.getOutputNeurons()) {
04            neuron.setError(outputError[i]);
05            double neuronError = neuron.getError();
06            // 根据所有的神经元输入迭代学习
07            for (Connection connection :neuron.getInputConnections()) {
08                 // 神经元的一个输入
09                 double input = connection.getInput();
10                 // 计算权值的变更
11                 double weightChange=  neuronError * input;
12                 // 更新权值
13                 Weight weight = connection.getWeight();
14                 weight.weightChange = weightChange;
15                 weight.value += weightChange;
```

```
16            }
17            i++;
18        }
19    }
```

第 11 行是算法的核心，它与感知机学习计算公式完全一致，即使用神经元误差和输入的乘积作为权重的调整值。

PerceptronLearningRule 的定义如下：

```
public class PerceptronLearningRule extends SupervisedLearning implements Serializable {
```

4.3.2 一个自学习的感知机实现——SimplePerceptron

我们已经具备构造一个具有学习能力的感知机的全部组件，剩下的工作就是将它们组合起来了。这里，我将这个感知机命名为 SimplePerceptron。大家可以在配套代码中查找 SimplePerceptron.java 文件来找到它。

SimplePerceptron 可以继承自 NeuralNetwork：

```
public class SimplePerceptron extends NeuralNetwork {
```

构造函数中的核心实现如下：

```
01 private void createNetwork(intinputNeuronsCount) {
02        // 设置网络类别为感知机
03        this.setNetworkType(NeuralNetworkType.PERCEPTRON);
04
05        // 输入神经元建立，表示输入的刺激
06        NeuronPropertiesinputNeuronProperties = new NeuronProperties();
07        inputNeuronProperties.setProperty("neuronType", InputNeuron.class);
08
09        // 由输入神经元构成的输入层
10        Layer inputLayer = LayerFactory.createLayer(inputNeuronsCount, inputNeuron
          Properties);
11        this.addLayer(inputLayer);
12        // 在输入层增加 BiasNeuron，表示神经元偏置
13        inputLayer.addNeuron(new BiasNeuron());
14        // 传输函数是 Step
15        NeuronPropertiesoutputNeuronProperties = new NeuronProperties();
16        outputNeuronProperties.setProperty("transferFunction", TransferFunction
          Type.STEP);
```

```
17
18        // 输出层，也就是神经元
19        Layer outputLayer = LayerFactory.createLayer(1, outputNeuronProperties);
20        this.addLayer(outputLayer);
21
22        // 将输入层和输出层进行全连接
23        ConnectionFactory.fullConnect(inputLayer, outputLayer);
24        NeuralNetworkFactory.setDefaultIO(this);
25        // 设置感知机学习算法
26        this.setLearningRule(new PerceptronLearningRule());
27 }
```

鉴于在代码中已经给出了详细的注释，这里不再详细介绍网络的构造过程。值得注意的两点是：第一，在第 23 行，进行了从输入层到输出层的全连接，连接的权重是一个随机值，不存在事先设计的情况；第二，在第 26 行，设置感知机的学习规则为 PerceptronLearningRule，这正是在上一节中已经实现的学习算法。

4.3.3　小试牛刀——SimplePerceptron 学习逻辑与

有了 SimplePerceptron，现在就让我们再次回到本章第一节“使用感知机记忆逻辑与”的内容。在第一节中，我们使用的是人为构造的感知机，事先设计感知机网络的权值，使得感知机可以正确处理逻辑与。但在本节中，将使用看似更先进的做法，即只给出学习数据，让感知机自行进行迭代训练，从而达到记忆的目的。本节的相关代码是 AndPerceptron。

首先，给出逻辑与学习的训练数据：

```
// 数据集有 2 个输入和 1 个输出
// 测试数据是 And 逻辑运行的结果
DataSettrainingSet = new DataSet(2, 1);
trainingSet.addRow(new DataSetRow(new double[]{0, 0}, new double[]{0}));
trainingSet.addRow(new DataSetRow(new double[]{0, 1}, new double[]{0}));
trainingSet.addRow(new DataSetRow(new double[]{1, 0}, new double[]{0}));
trainingSet.addRow(new DataSetRow(new double[]{1, 1}, new double[]{1}));
```

上述代码中，DateSetRow 的构造函数接受两个参数，第一个为输入向量，第二个为期望值。这些数据用于训练神经网络。

接着创建一个只有两个输入节点的感知机：

```
SimplePerceptronmyPerceptron = new SimplePerceptron(2);
```

使用训练数据训练感知机：

```
myPerceptron.learn(trainingSet);
```

训练完成后，即可对网络进行测试，如下所示：

```
01 public static void testNeuralNetwork(NeuralNetworkneuralNet, DataSettestSet) {
02      for(DataSetRowtestSetRow : testSet.getRows()) {
03           neuralNet.setInput(testSetRow.getInput());
04           neuralNet.calculate();
05           double[] networkOutput = neuralNet.getOutput();
06
07           System.out.print("Input: " + Arrays.toString( testSetRow.getInput() ) );
08           System.out.println("Output: " + Arrays.toString( networkOutput) );
09      }
10 }
```

上述代码用于测试感知机，第 3 行用于设置网络的输入，第 4 行进行计算，第 5 行得到网络的输出，第 7～8 行进行输出的打印。

执行 AndPerceptron 后，一种可能的输出如下（限于篇幅只截取部分）：

```
Training neural network...
iterate:1
.........
iterate:6
Testing trained neural network
Input: [0.0, 0.0]Output: [0.0]
Input: [0.0, 1.0]Output: [0.0]
Input: [1.0, 0.0]Output: [0.0]
Input: [1.0, 1.0]Output: [1.0]
```

可以看到，在进行 6 次迭代训练后，网络学习结束。在测试过程中，给出四组数据进行逻辑与运算，网络均给出了正确的结果。

4.3.4 训练何时停止

虽然到目前为止，我们已经完整实现了一个感知机，并且通过这个感知机已经可以进行简单的学习了。但细心的读者不难发现，在整个训练过程中，似乎缺少了一个关键环节的解释，即：作为一种迭代训练，那它在什么条件下停止训练呢？

要解答这个问题，我们需要更进一步查看下面代码的具体实现：

```
myPerceptron.learn(trainingSet);
```

上述代码给出一组训练数据并让神经网络学习。但神经网络将学习逻辑完全委托给学习规则进行：

```
learningRule.learn(trainingSet);
```

对于迭代学习（大部分神经网络使用这种学习方式）来说，其实现如下：

```
01 final public void learn(DataSettrainingSet) {
02       setTrainingSet(trainingSet); // set this field here su subclasses can access
         it
03       onStart();
04
05       while (!isStopped()) {
06           beforeEpoch();
07           //训练网络
08           doLearningEpoch(trainingSet);
09           this.currentIteration++;
10           afterEpoch();
11
12           // 是否达到停止的条件，如误差在可接受范围内，
13           // 或者达到最大迭代次数
14           if (hasReachedStopCondition()) {
15                stopLearning();
16           } else if (!iterationsLimited&& (currentIteration == Integer.MAX_VALUE))
             {
17                // 达到最大迭代次数并且迭代次数不是无限的，重置计数器
18                this.currentIteration = 1;
19           }
20
21           // 触发一次学习事件
22           fireLearningEvent(new LearningEvent(this));
23
24           ...省略部分不相干代码.....
25       }
26       // 触发一次学习结束事件
27       fireLearningEvent(new LearningStoppedEvent(this));
28 }
```

代码第 5 行，一次学习周期正式开始。在这里，每一次数据迭代称为一个纪元（Epoch）。一个纪元的学习，包括对给出的训练数据的一次完整遍历。一个纪元的学习完成后，如果

没有达到终止条件，则会对训练数据进行第二次甚至更多次的训练。

第 12～19 行判断是否达到训练终止条件。如果达到，则终止循环，结束训练；如果没有达到终止条件，则继续训练。终止条件封装在 StopCondition 接口中：

```
public interface StopCondition {
    publicbooleanisReached();
}
```

第 14 行的 hasReachedStopCondition()函数会遍历学习算法中所有的终止条件 StopCondition，以确定是否可以结束学习过程。常用的终止条件有：

- MaxErrorStop：实际的产生的误差小于期望值，则停止学习。
- MaxIterationsStop：学习的迭代次数（指 Epoch 次数）达到预设值。
- SmallErrorChangeStop：多个 Epoch 周期的学习对误差的修正已经不大，则终止学习。

在本例中，使用的策略是 MaxErrorStop，即当整体误差小于预设值时，则终止学习。默认情况下，该阈值为 0.01。

根据不同的学习算法和特点，完全可以自定义终止条件，以满足不同的需求。

4.4 再看坐标点位置识别

在本章前文中，我们曾经自行设计了一个识别坐标系统坐标点所在象限的感知机。在本节中，我们将使用感知机学习规则，重新实现该功能。不同之处是，在这里，我们将仅给出训练数据，让感知机自行学习，调整其神经元权重值。

为了让训练取得较好的效果，这里使用 40000 条数据进行学习，每个象限分配到 10000 条数据。数据由随机函数产生，所有数据均匀分布在 0 和 1 之间（对应代码参见配套代码中的 PerceptronAxisClassify 类）：

```
for (int i = 0; i < 10000; i++) {
        // 第一象限
        td.addRow(new DataSetRow(new double[] { 1 * nextDouble(), 1 * nextDouble() },
        new double[] { 1, 1 }));
        // 第二象限
```

```
        td.addRow(new DataSetRow(new double[] { -1 * nextDouble(), 1 * nextDouble() },
        new double[] { 0, 1 }));
        // 第三象限
        td.addRow(new DataSetRow(new double[] { -1 * nextDouble(), -1 * nextDouble() },
        new double[] { 0, 0 }));
        // 第四象限
        td.addRow(new DataSetRow(new double[] { 1 * nextDouble(), -1 * nextDouble() },
        new double[] { 1, 0 }));
}
```

这里依然使用[1 1]、[0 1]、[0 0]、[1 0]分别表示第一、二、三、四共 4 个象限。其中，nextDouble()函数返回一个介于 0 和 1 之间的非 0 浮点数。

接着构造感知机，并进行训练：

```
SimplePerceptron2 myPerceptron = new SimplePerceptron2(2);
PerceptronLearningRulelearningRule=(PerceptronLearningRule) myPerceptron.getLearningRule();
learningRule.setMaxError(0.001);
myPerceptron.learn(trainingSet);
```

为了减少误差，这里将最大可接受误差设置为 0.001。这样，网络就会进行更多次的迭代，以满足较小的误差需求。类名 SimplePerceptron2，表示有两个输出神经元的感知机；构造函数参数中的 2，表示输入向量长度为 2。

训练完成后，进行测试，随机选取若干个坐标点，让感知机进行判断：

```
DataSet td = new DataSet(2, 2);
for (int i = 0; i < 1000; i++) {
        // 第一象限
        td.addRow(new DataSetRow(new double[] { 1 * nextDouble(), 1 * nextDouble() },
        new double[] { 1, 1 }));
        // 第二象限
        td.addRow(new DataSetRow(new double[] { -1 * nextDouble(), 1 * nextDouble() },
        new double[] { 0, 1 }));
        // 第三象限
        td.addRow(new DataSetRow(new double[] { -1 * nextDouble(), -1 * nextDouble() },
        new double[] { 0, 0 }));
        // 第四象限
        td.addRow(new DataSetRow(new double[] { 1 * nextDouble(), -1 * nextDouble() },
        new double[] { 1, 0 }));
}
```

这里给出了期望输出，用于检查感知机的工作效果，统计正确率。下述代码根据期望

输出计算感知机判断的正确率：

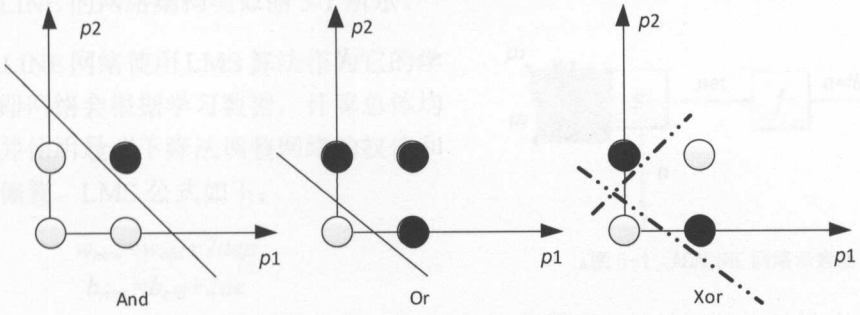
```
01 intcorrectCount = 0;
02 intincorrectCount = 0;
03 for (DataSetRowtestSetRow :td.getRows()) {
04    neuralNet.setInput((testSetRow.getInput()));
05    neuralNet.calculate();
06    double[] networkOutput = neuralNet.getOutput();
07    if (Arrays.equals(networkOutput, testSetRow.getDesiredOutput())) {
08          correctCount++;
09    } else {
10          incorrectCount++;
11    }
12 }
13 System.out.println("正确率: "+correctCount * 1.0 / (correctCount + incorrectCount));
```

第 7 行代码将网络的实际输出和期望输出进行比较，判断网络的输出结果是否正确，同时统计判断正确的次数。

执行上述代码，待网络训练完成后就会进行测试，最终的正确率保持在 99%以上。

4.5 感知机的极限——异或问题

到目前为止，我们的感知机工作状态良好，但这是否意味着我们找到了一个全能的神经网络，它能解决一些问题呢？很不幸，答案是否定的。根据前面的描述，感知机是依靠判定边界进行数据分类的，如果数据分布无法通过简单的平面划分，那么感知机就无法做出正确的响应。图 4-3 所示显示了逻辑与、逻辑或和异或运算的数据分类情况。



▲图 4-3 逻辑与、逻辑或和异或运算的数据分类情况

可以很明显地看到，对于 And 和 Or 运算，可以简单地使用一个判定平面划分两种不同的类别；对于异或操作，则必须使用两个判定边界。换言之，单层感知机无法处理这种线性不可分问题。

我们可以使用 SimplePerceptron 感知机来验证这个问题。异或问题的训练数据如下：

```
DataSettrainingSet = new DataSet(2, 1);
trainingSet.addRow(new DataSetRow(new double[]{0, 0}, new double[]{0}));
trainingSet.addRow(new DataSetRow(new double[]{0, 1}, new double[]{1}));
trainingSet.addRow(new DataSetRow(new double[]{1, 0}, new double[]{1}));
trainingSet.addRow(new DataSetRow(new double[]{1, 1}, new double[]{0}));
```

当使用这些数据训练 SimplePerceptron 时，不难发现，系统永远无法将误差控制在令人满意的范围，因此训练过程会毫无休止地不断执行，整个训练最终宣告失败。

4.6 总结

本章详细介绍了使用 Neuroph 实现感知机及其标准学习规则的完整过程，并且使用该感知机处理了逻辑与和坐标点识别两个问题。在最后，也提出了感知机的局限性，即无法处理非线性问题。对于这个缺陷，将在后续的章节中详细讨论解决方案。

第5章 ADALINE网络及其应用

ADALINE是一种与感知机非常类似的神经网络。本章将重点介绍这个网络，并给出ADALINE网络的一种实现。ADALINE网络使用的LMS学习算法也将在本章进行简单介绍。本章的最后还将给出一个使用ADALINE网络进行数字识别的案例。在这个案例中，你会发现ADALINE网络不仅可以正确识别训练数据，而且具备一定的抗干扰能力，可以对含有噪点的数据进行处理。

5.1 ADALINE网络与LMS算法

ADALINE是1960年由Widrow和他的研究生提出的，全称是ADAptiveLInearNEuron，即自适应线性神经元。与感知机不同的是，ADALINE网络使用了线性函数*Linear*作为传输函数，而不是*Step*函数。

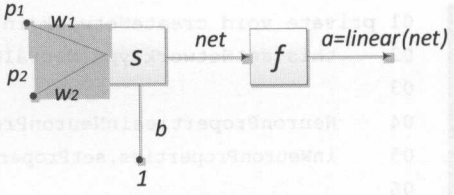
因此网络的输出a为：

$$a = Linear(Wp+b)=Wp+b$$

ADALINE的网络结构类似图5-1所示。

ADALINE网络使用LMS算法作为它的学习算法，即网络会根据学习数据，计算总体均方误差，并使用最速下降法调整网络的权值和神经元的偏置。LMS公式如下：

$$w_{new}=w_{old}+2aep$$
$$b_{new}=b_{old}+2ae$$



▲图5-1 ADALINE网络示意图

上述两式分别描述了每次迭代训练中权值和偏置的修正算法。细心的读者应该会发

现，它们与感知机学习算法十分相似。为方便读者比较，这里再次给出感知机学习公式：

$$w_{new}=w_{old}+ep$$
$$b_{new}=b_{old}+e$$

从形式上看，两者似乎只相差了误差系数 2α。但本质上来说，两者具有以下不同点。

（1）推导过程完全不同。感知机规则通过作图法对判断边界进行分析，最终通过分情况讨论并归纳整理公式得到。而 LMS 算法则是通过计算最小均方差并利用最速下降法求得。

（2）感知机学习规则中误差 e 是离散值，因为传输函数 *step* 只给出离散输出。而 LMS 规则中，误差 e 是连续的。

（3）LMS 算法中存在学习步长系数 α，它是由最速下降法引入的，表示学习的速度。而感知机规则中没有此项。

对 LMS 算法和感知机算法的推导有兴趣的读者可以参考《神经网络设计》一书。

对于系数 α，它表示学习的速度或者步长。需要取一个合理的数值，一般是在 0.1 或 0.01 这样的数量级。如果步长太大，那么学习过程就不会特别精准，网络极有可能错过最优解，无法取得满意的结果。反之，如果步长太小，那么学习速度就会变慢，需要更多的迭代次数来达到满足要求的学习误差，且更容易陷入局部最优。

5.2 ADALINE 网络的 Java 实现

有了感知机网络作为铺垫，我们就可以很容易地实现 ADALINE 网络。与感知机最大的不同，仅仅只是输出层的传输函数不同而已。整体网络结构都是类似的，因此其实现的核心代码如下：

```
01 private void createNetwork(intinputNeuronsCount, intoutputNeuronsCount) {
02     this.setNetworkType(NeuralNetworkType.ADALINE);
03
04     NeuronPropertiesinNeuronProperties = new NeuronProperties();
05     inNeuronProperties.setProperty("transferFunction", TransferFunctionType.LINEAR);
06
07     Layer inputLayer = LayerFactory.createLayer(inputNeuronsCount, inNeuronProperties);
08     inputLayer.addNeuron(new BiasNeuron());
09
```

```
10    this.addLayer(inputLayer);
11
12    NeuronPropertiesoutNeuronProperties = new NeuronProperties();
13     outNeuronProperties.setProperty("transferFunction", TransferFunctionType.LINEAR);
14
15    Layer outputLayer = LayerFactory.createLayer(outputNeuronsCount, outNeuronProperties);
16    this.addLayer(outputLayer);
17
18    ConnectionFactory.fullConnect(inputLayer, outputLayer);
19
20    NeuralNetworkFactory.setDefaultIO(this);
21
22    LMS lms=new LMS();
23    lms.setLearningRate(0.05);
24    lms.setMaxError(0.5);
25    this.setLearningRule(lms);
26 }
```

第 7～8 行创建输入层，并定义贝叶斯神经元。第 12～13 行定义了输出层的传输函数为线性函数，这也正是与感知机最大的不同之处。第 18 行将输入层和传输层进行全连接。第 22～25 行指定 ADALINE 网络的学习算法为 LMS 算法，并设置学习步长为 0.05，最大可接受误差为 0.5。

学习步长和最大误差的设置原则为：尽可能使得网络在更少的迭代次数下，取得最小的误差。同时，在网络误差可以满足应用要求的前提下，设置一个最大的误差，从而避免进行更长时间的学习。

学习算法 LMS 的核心代码如下：

```
01 protected void updateNetworkWeights(double[] outputError) {
02       int i = 0;
03       // for each neuron in output layer
04       for (Neuron neuron :neuralNetwork.getOutputNeurons()) {
05             neuron.setError(outputError[i]);
06             this.updateNeuronWeights(neuron);
07             i++;
08       }
09 }
10
11 protected void updateNeuronWeights(Neuron neuron) {
12       // 取得神经元误差
13       double neuronError = neuron.getError();
14
```

```
15      // 根据所有的神经元输入迭代学习
16      for (Connection connection :neuron.getInputConnections()) {
17          // 神经元的一个输入
18          double input = connection.getInput();
19          // 计算权值的变更
20          double weightChange = this.learningRate * neuronError * input;
21          // 更新权值
22          Weight weight = connection.getWeight();
23          weight.weightChange = weightChange;
24          weight.value += weightChange;
25      }
26  }
```

其中 updateNetworkWeights()函数为神经网络学习算法的核心函数，被 Neuroph 框架回调。传入参数 outputError 为一次迭代学习的实际误差，即期望值与实际值之间的差值。网络根据这个误差不断地调整其神经元权值。

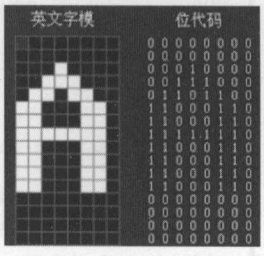
第 17～24 行是根据每个神经元的误差信息，进行每个连接权重的调整。第 20 行使用的公式也正是 LMS 算法的核心公式，其中 learningRate 即学习步长 2α，因为 2 为常数，没有实际存在价值，所以在实现中将其省略。

5.3 使用 ADALINE 网络识别数字

ADALINE 网络看似简单，但已经可以做一些有趣的事情了。在本节中，将使用 ADALINE 网络进行印刷体数字识别。在这里将会详细阐述解决该问题的思想方法和具体实现步骤。

5.3.1 印刷体数字识别问题概述

众所周知，表示字体或者图像最简单的方式即是使用点阵字体。在点阵字体中，一个文字被编码成一个有着行和列信息的二维数组。对于二值图像而言，如果背景为黑色，字体为白色，那么可以使用 0 表示黑色，1 表示白色，使用点阵图像来勾画整个文字，如图 5-2 所示。



▲图 5-2　点阵字体的表示

右边的二维数组即为点阵字体 A 的内部表示。以此类推，如果要表示数字 0～9，则可以使用同样的方法画出字体，然后进行

编码，将每个数字表示为一个二维数组即可。而这个二维数组便可以作为 ADALINE 网络的输入。

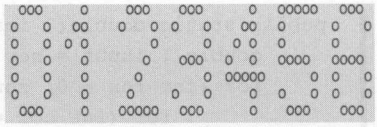
对于数字识别问题来说，可能的答案只有 10 个，即 0～9，因此网络的输出可以是一个 10 维向量，向量上的每一个分量分别对应一个数字。

这样，ADALINE 网络的输入和输出都有了明确的定义，就可以开始动手实现了！

5.3.2 代码实现

为方便描述，在代码实现中，采用 5×7 的点阵大小来表示每一个数字字符。图 5-3 所示显示了本次训练中的部分点阵数字。

每个字符宽度为 5 个像素，高度为 7 个像素，因此 ADALINE 的输入神经元个数为 5×7=35 个。



▲图 5-3 ADALINE 训练用点阵数字

5.3.2.1 数字定义

限于篇幅，这里只给出部分数字的定义（读者可以参考随书代码中的 AdalineDemo 类查看完整的代码）：

```
public final static int CHAR_WIDTH = 5;
public final static int CHAR_HEIGHT = 7;
public static String[][] DIGITS = {
  {
    " ooo ",
    "o   o",
    "o   o",
    "o   o",
    "o   o",
    "o   o",
    " ooo " },

  {
    "  o  ",
    " oo  ",
    "o o  ",
    "  o  ",
    "  o  ",
    "  o  ",
    "  o  " },
    省略其他数字定义
```

5.3.2.2　定义训练数据集与 ADALINE 网络

这里所有的数字存放在二维数组 DIGITS 中。接着可以定义 ADALINE 网络和学习数据集：

```
Adalineada = new Adaline(CHAR_WIDTH * CHAR_HEIGHT, DIGITS.length);
DataSet ds = new DataSet(CHAR_WIDTH * CHAR_HEIGHT, DIGITS.length);
```

上述代码定义 ADALINE 网络的输入节点为 35 个，输出节点为 10 个，数据集大小和网络规模完全一致。

将二维数组 DIGITS 转为网络可以识别的格式，即转为训练数据。这里将背景使用-1 表示，字体内容使用数字 1 表示：

```
public static double[] image2data(String[] image) {
     double[] input = new double[CHAR_WIDTH * CHAR_HEIGHT];
     for (int row = 0; row < CHAR_HEIGHT; row++) {
          for (int col = 0; col < CHAR_WIDTH; col++) {
               int index = (row * CHAR_WIDTH) + col;
               charch = image[row].charAt(col);
               input[index] = ch == 'O' ? 1 : -1;
          }
     }
     return input;
}
```

有了 image2data()函数，就将一个点阵数字转为标准的神经网络训练数据 double 数组。最后，需要将这些训练数据添加到训练数据集中：

```
01 public static DataSetRowcreateTrainDataRow(String[] image, intidealValue) {
02   double[] output = new double[DIGITS.length];
03   for (int i = 0; i <output.length; i++)
04          output[i] = -1;
05
06   double[] input = image2data(image);
07
08   output[idealValue] = 1;
09   DataSetRowdsr = new DataSetRow(input, output);
10   return dsr;
11 }
12
13 for (int i = 0; i <DIGITS.length; i++) {
14   ds.addRow(createTrainDataRow(DIGITS[i], i));
15 }
```

上述代码的 createTrainDataRow()函数将点阵字体转为 DataSetRow，第 6 行调用了

image2data()函数；参数 idealValue 表示这条训练数据的期望输出。

第 13～15 行将所有的训练数据加入训练集。

5.3.2.3 使用数据集训练网络

接着，即可使用一句代码训练 ADALINE 网络：

```
ada.learn(ds);
```

这里，ADALINE 网络就会使用前文中定义的 LMS 算法进行学习。

5.3.2.4 验证网络性能

训练结束后，测试网络，验证其学习效果：

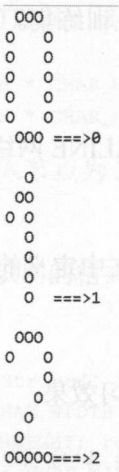
```
1 for (int i = 0; i <DIGITS.length; i++) {
2    ada.setInput(image2data(DIGITS[i]));
3    ada.calculate();
4    printDigit(DIGITS[i]);
5    System.out.println(maxIndex(ada.getOutput()));
6    System.out.println();
7 }
```

上述代码第 2 行设置网络的输入，即点阵数字图像。第 3 行进行网络计算进行识别。第 4 行打印数字图像。第 5 行打印网络的识别结果。需要注意的是，由于网络的输出是 double 数组，那么我们怎么确定这个 double 数组究竟是表示哪个数字呢？规则很简单，在训练的时候，训练数据总是将表示这个数字的那个向量的维度设为 1，其他均设置为 0。比如，当表示数字 4 的时候，数组的第 4 个索引下标为 1，其余 9 个均为 0。因此，当给出任意一个 10 维 double 数组时，我们可以采用竞争规则，将最大的那个维度视为 1，其余都视为 0。而这正是 maxIndex()函数的作用，即找到数组中的最大值的索引下标：

```
1 public static intmaxIndex(double[] data) {
2    int result = -1;
3    for (int i = 0; i <data.length; i++) {
4          if (result == -1 || data[i] >data[result]) {
5                result = i;
6          }
7    }
8    return result;
9 }
```

执行上述所有代码，就可以看到网络对印刷体的识别结果。部分输出如图 5-4 所示，

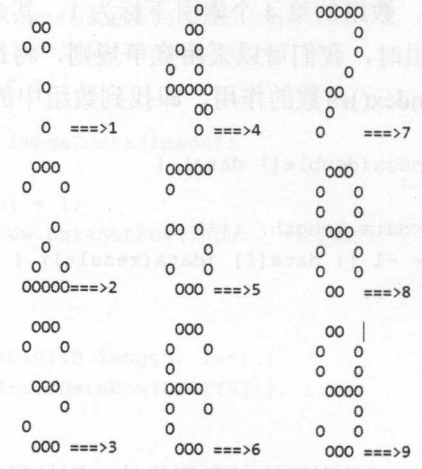
左边表示输入数字图像，右边表示识别结果。执行几次，不难发现其识别正确率达到 100%。



▲图 5-4　ADALINE 数字识别部分输出

5.3.3　加入噪点后再尝试

ADALINE 网络的强大之处并不仅仅在于它可以识别已经训练过的数据，即使在数据中加入一些噪点，ADALINE 网络依然有很大的概率可以将其识别正确。图 5-5 所示显示了 ADALINE 网络在进行标准的数字训练后，再进行带有噪点的字符识别的执行结果。不难发现，在本次验证中，所有字符都与训练数据存在差异，或者残缺不全，或者存在多余噪点，但这都不影响 ADALINE 网络正确识别这些数字。



▲图 5-5　增加噪点后的识别结果

有兴趣的读者可以参考随书代码中的 AdalineDemo2 类。

5.4 总结

本章主要介绍了 ADALINE 网络的结构、原理和学习算法 LMS。ADALINE 网络的结构和感知机大体相似，但是其主要的传输函数是使用线性函数而非 *step* 函数。此外，尽管 LMS 算法的公式和感知机学习规则非常相似，但是它们两者的原理和推导过程是截然不同的。本章还重点介绍了如何使用 ADALINE 网络进行印刷体的数字识别。从实验效果看，ADALINE 网络不仅可以识别标准的训练数据，而且具有一定的抗干扰能力。

第6章　多层感知机和BP学习算法

到目前为止，我们已经了解了感知机以及 ADALINE 网络这两种最为简单的神经网络。本章将对单层感知机网络进行扩展，介绍一种多层感知机结构，也就是多层感知机神经网络。同时，还将介绍适用于这种多层感知机的 BP 学习算法。多层感知机与 BP 学习算法共同构成了被大家所熟知的，也是目前使用最为广泛的一种神经网络——BP 神经网络。通过本章的学习，你将知道：

- 多层感知机的结构；
- BP 学习算法的应用；
- 通过 Java 构造一个 BP 神经网络。

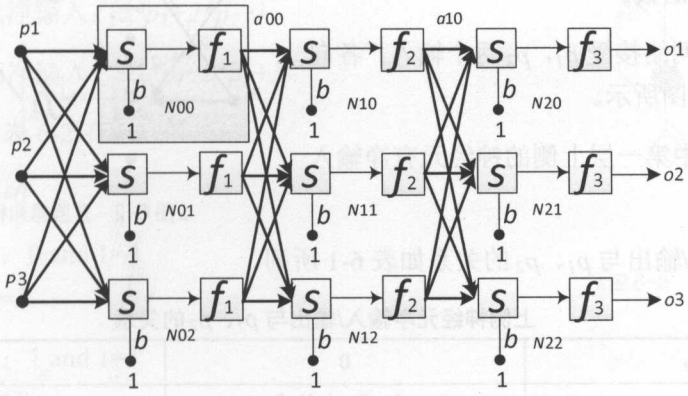
6.1 多层感知机的结构与简单实现

6.1.1 多层感知机结构的提出

简单的感知机无法处理非线性问题，这是感知机使用的巨大局限性，这一问题也一度造成神经网络长达 10 年的发展停滞。为了克服这类问题，Widrow 提出了一种新的方法：多层感知机。顾名思义，多层感知机叠加复合使用了多个感知机，通过某种手段，将多个感知机联合使用，使之作为一个整体工作，配合合适的算法，就可以用来处理非线性问题。多层感知机的基本结构如图 6-1 所示。

该多层感知机有 3 层组成，每层各 3 个神经元。该多层感知机接受 3 个输入，并产生 3 个输出。多层感知机的每个感知机的工作方式和单个感知机是一样的，它们都接受若干个输入，并通过传输函数给出一个输出。由图 6-1 中可以看到，该网络接受 $p1$、$p2$、$p3$

三个输入，经过网络计算后，给出 $o1$、$o2$、$o3$ 三个输出。



▲图 6-1 多层感知机结构

为图中的每一个神经元进行编号，从左上角开始，记为 $N00$，第一层依次记为 $N00$、$N01$、$N02$。同理，第二层各神经元记为 $N10$、$N11$、$N12$；第三层各神经元记为 $N20$、$N21$、$N22$。如图 6-1 所示，$N00$、$N01$、$N02$ 分别依次接受 $p1$、$p2$ 和 $p3$ 作为输入，通过其传输函数计算，给出 $a00$、$a01$ 和 $a02$ 三个输出。记 $N00$ 和 $N10$ 的连接权值为 W_{0010}，并依次类推记 $N01$ 和 $N11$ 的权值为 W_{0111}……

则 $N10$ 的净输入为：

$$a0*W_{0010}+a01*W_{0110}+a02*W_{0210}$$

$N11$ 的净输入为：

$$a0*W_{0011}+a01*W_{0111}+a02*W_{0211}$$

$N12$ 的净输入为：

$$a0*W_{0012}+a01*W_{0112}+a02*W_{0212}$$

同理，每一个神经元的净输入都是前一层神经元输出到当前神经元的加权和（即神经元的输入函数为加权求和函数）。通过这种方式，网络的输入从第一层神经元一直传递到最后一层，并由最后一层神经元给出该网络的输出。同时，每一层神经元的传输函数可以是相同的，也可以是不同的。但同一层神经元中，一般来说，传输函数保持一致。

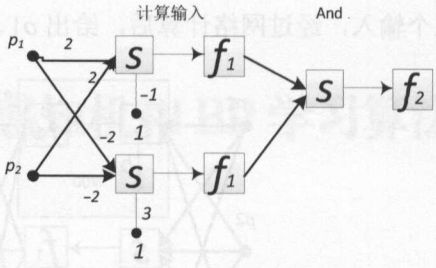
6.1.2 定义多层感知机处理异或问题

利用该结构，连接多个神经元就可以处理异或问题。简单地使用一个两层神经网络就

可以记忆异或运算。一个可行的方案如图 6-2 所示，f_1 使用 *Step* 函数。

图 6-2 中，网络接受 p_1，p_2 两个输入，各向权值和偏置也如图所示。

对于图 6-2 中第一层上侧的神经元有净输入 $2*p_1+2*p_2-1$。



▲图 6-2　多层感知机处理异或

它的净输入/输出与 p_1、p_2 的关系如表 6-1 所列。

表 6-1　　上侧神经元净输入/输出与 p_1、p_2 的关系

p_1/p_2	0	1
0	−1　*Step*(−1)=0	1　*Step*(1)=1
1	1　*Step*(1)=1	3　*Step*(3)=1

实际上，这个神经元对输出结果进行了图 6-3 所示的划分。

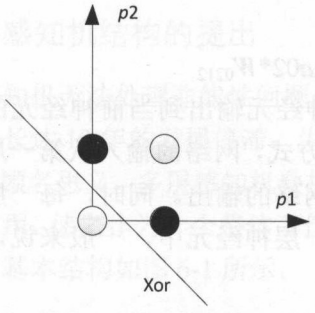
对于图 6-2 中第一层下侧的神经元有净输入 $-2*p_1-2*p_2+3$。

它的净输入/输出与 p_1、p_2 的关系如表 6-2 所列。

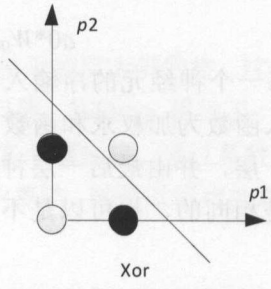
表 6-2　　下侧净输入/输出与 p_1、p_2 的关系

p_1/p_2	0	1
0	3　*Step*(3)=1	1　*Step*(1)=1
1	1　*Step*(1)=1	−1　*Step*(−1)=0

因此，下侧神经元对数据进行了图 6-4 所示的划分。



▲图 6-3　上侧神经元的划分



▲图 6-4　下侧神经元的划分

最后，由输出神经元对两个神经元的数据进行整合，这里使用逻辑与操作。最终可以得到图 6-5 所示的正确的异或运算的划分。

具体的工作过程如下。

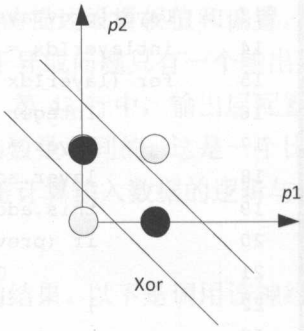
上侧神经元净输入：$2*p_1+2*p_2-1$

下侧神经元净输入：$-2*p_1-2*p_2+3$

查表 6-1 和表 6-2 有：

- $p_1=0\ p_2=0$
  - 输出：0 and 1=0
- $p_1=0\ p_2=1$
  - 输出：1 and 1=1
- $p_1=1\ p_2=0$
  - 输出：1 and 1=1
- $p_1=1\ p_2=1$
  - 输出：1 and 0=0

不难看出，该结构的网络已经可以正确处理异或。



▲图 6-5 最终的异或运算划分

6.1.3 多层感知机的简单实现

上一节详细介绍了多层感知机的工作原理，并使用多层感知机成功解决了异或问题。但上一节的内容仅停留在理论阶段，本节将根据上一节多层感知机的原理，基于 Neuroph 框架给出一个合理的多层感知机实现。读者可以参考随书代码 MlPerceptronAndOutputNoLearn：

```
01 protected void createNetwork(List<Integer>neuronsInLayers, NeuronPropertiesneuron
   Properties) {
02     //多层感知机
03     this.setNetworkType(NeuralNetworkType.MULTI_LAYER_PERCEPTRON);
04
05     // 输入层，代表每一个输入数据
06 NeuronPropertiesinputNeuronProperties =
07 new NeuronProperties(InputNeuron.class, Linear.class);
08     Layer layer = LayerFactory.createLayer(neuronsInLayers.get(0), inputNeuron
       Properties);
09     layer.addNeuron(new BiasNeuron());
10     this.addLayer(layer);
11
12     // 建立中间隐层
```

```
13      Layer prevLayer = layer;
14      intlayerIdx = 1;
15      for (layerIdx = 1; layerIdx<neuronsInLayers.size()-1; layerIdx++) {
16            Integer neuronsNum = neuronsInLayers.get(layerIdx);
17            layer = LayerFactory.createLayer(neuronsNum, neuronProperties);
18            layer.addNeuron(new BiasNeuron());
19            this.addLayer(layer);
20            if (prevLayer != null) {
21                  ConnectionFactory.fullConnect(prevLayer, layer);
22            }
23            prevLayer = layer;
24      }
25
26      //设置初始权值
27      Neuron n1=layer.getNeuronAt(0);
28      Connection[] c1=n1.getInputConnections();
29      c1[0].setWeight(new Weight(2));
30      c1[1].setWeight(new Weight(2));
31      c1[2].setWeight(new Weight(-1));
32
33      Neuron n2=layer.getNeuronAt(1);
34      Connection[] c2=n2.getInputConnections();
35      c2[0].setWeight(new Weight(-2));
36      c2[1].setWeight(new Weight(-2));
37      c2[2].setWeight(new Weight(3));
38
39      // 建立输出层
40      Integer neuronsNum = neuronsInLayers.get(layerIdx);
41      NeuronPropertiesoutProperties=new NeuronProperties();
42      // 输入函数为逻辑与
43      outProperties.put("inputFunction", org.neuroph.core.input.And.class);
44      layer = LayerFactory.createLayer(neuronsNum, outProperties);
45      this.addLayer(layer);
46      ConnectionFactory.fullConnect(prevLayer, layer);
47      prevLayer = layer;
48
49      NeuralNetworkFactory.setDefaultIO(this);
50  }
```

首先，感知机需要一个输入层。输入层神经元的数量和输入参数数量保持一致，即每一个输入神经元都接受一个参数。为了给中间层的神经元设置偏置，在第 9 行输入层加入一个额外的偏置神经元，用于模拟下一层所有神经元的偏置值。多层神经元可以有多个中间层，数量不限，但是一般 1～3 层应该足够解决大部分问题。在建立中间层神经元后，

在 26～37 行设置神经元间的初始连接权值。这里，根据图 6-2 的描述设置权值和偏置，以确保神经网络可以正确记忆异或问题。最后，建立输出层。由于异或问题只有一个输出，所以输出层只有一个神经元。这里一个特别值得注意的地方是，第 43 行中，输出层配置了逻辑与作为其输入函数。这与绝大多数时候使用的加权求和函数是不同的。这是一个比较特别的输入函数，它表示神经元的净输入不再是加权求和，而是计算输入数据的逻辑与，并将结果作为净输入。

该多层感知机将按照上一节所述的方式运行，并给出正确的结果。以下是调用该神经工作的代码（参考随书代码 XorMlPerceptronAndSimple）：

```
// 建立异或测试数据，不需要训练神经网络，故无需期望值
DataSettrainingSet = new DataSet(2,1);
trainingSet.addRow(new DataSetRow(new double[]{0, 0}, new double[]{Double.NaN}));
trainingSet.addRow(new DataSetRow(new double[]{0, 1}, new double[]{Double.NaN}));
trainingSet.addRow(new DataSetRow(new double[]{1, 0}, new double[]{Double.NaN}));
trainingSet.addRow(new DataSetRow(new double[]{1, 1}, new double[]{Double.NaN}));

// 神经网络使用 Step 作为传输函数
// 拥有 2 个输入点、2 个中间神经元和 1 个输入神经元
MlPerceptronmyPerceptron = new
MlPerceptronAndOutputNoLearn(TransferFunctionType.STEP,2,2,1);

//测试网络对异或的反应
for(DataSetRowtestSetRow : trainingSet.getRows()) {
    myPerceptron.setInput(testSetRow.getInput());
    myPerceptron.calculate();
    double[] networkOutput = myPerceptron.getOutput();

    System.out.print("Input: " + Arrays.toString( testSetRow.getInput() ) );
    System.out.println(" Output: " + Arrays.toString( networkOutput) );
}
```

首先给出 4 个输入，由于没有训练过程，因此不需要设定期望输出。设置神经网络的结构与图 6-2 一致，2 个输入点、2 个中间层神经元和 1 个输出神经元，并使用 *step* 函数作为传输函数。对每一条输入数据进行计算，得出：

```
Input: [0.0, 0.0] Output: [0.0]
Input: [0.0, 1.0] Output: [1.0]
Input: [1.0, 0.0] Output: [1.0]
Input: [1.0, 1.0] Output: [0.0]
```

从输出结果可以看到，该神经网络已经对异或运算进行了正确处理。

至此，多层神经元网络的工作原理已经基本介绍完毕，但对多层神经网络的学习还远远没有结束。其中，最为重要的多层神经网络的学习算法还没有介绍，在下一节中将重点介绍适用于多层感知机的学习算法——BP 算法。

6.2 多层感知机学习算法——BP 学习算法

到目前为止，已经详细介绍了一个简单的多层感知机的实现过程，并使用该感知机学习了异或运算。但细心的读者一定会发现，该神经网络模型中，尚有一个重要的疑问没有解决，那就是如何设定神经网络的各向权值和偏置。实际上，在一个实际使用的神经网络中，各向权值是很难人工设置的。也就是说，该神经网络并不具备学习能力。而学习能力恰恰是神经网络必须具备的重要特性。在本节中，将详细介绍一个典型的多层感知机学习算法——反向传播算法（Backpropagation，即 BP 学习算法）。

6.2.1 BP 学习算法理论介绍

多层感知机的学习算法的推导过程非常复杂，涉及线性代数、偏导数等数学知识，要求读者有良好的数学功底。本书将极尽可能避免复杂公式的出现，尽量使用较通俗的定性描述或初等数学替代严格的高等数学描述，力求读者能理解算法的基本思想。如果读者希望看到详细的推导公式，可以查阅相关专业书籍。

首先，请读者回顾一下神经网络模型图 6-1。这里，假设所有的输入函数都是加权求和函数。在实际使用中，使用最为广泛并具有一般意义的输入函数也正是加权求和函数。这与图 6-2 中给出的逻辑与函数是完全不同的。图 6-2 中介绍的逻辑与输入函数只是为了说明的便利和人工计算的方便，但缺乏普遍适用性，所以在本章的学习算法中，将统一使用加权求和函数作为每一个有效神经元的输入函数。同时，也将统一使用 S 型函数 *Sigmoid* 作为中间层神经元的传输函数。

神经网络学习算法的目的，是最终产生一个神经网络模型，对于这个模型，只要给定一个输入，可以给出一个期望的输出值。根据前几章对神经网络的描述，可以知道，神经网络的输出与输入值、神经元连接权重、偏置、传输函数、输入函数相关，但输入函数和传输函数在神经网络建立后，就不再改变，因此神经网络的学习过程，也就是不断地对神经元连接权重和偏置进行调整的过程。与感知机的学习规则类似，神经网络的权值调整也

需要有一个标准，该标准就是给定输入的期望输出，如果神经网络的实际输出与期望输出是一致的，那么可以认为该神经网络已经可以正常工作，反之，则需要根据期望输出与实际输出之间的误差来调整网络的各向权值。在简单的感知机学习规则中，由于只有一层网络，因此通过误差可以很容易地进行权值调整，在多层感知机网络中，这种方法同样适用，唯一不同的是，误差要通过最后一层网络，逐层向前传播，用于调衡各层的权值连接。

为了能更清楚地说明问题，进行如下定义：

- 神经网络输入：x_1，x_2，x_3，…，x_n
- 中间隐层净输入：hi_1，hi_2，hi_3，…，hi_p
- 中间隐层输出：ho_1，ho_2，ho_3，…，ho_p
- 输出层净输入：ai_1，ai_2，ai_3，…，ai_q
- 输出层输出：ao_1，ao_2，ao_3，…，ao_q
- 期望输出：t_1，t_2，t_3，…，t_q
- 输入层到中间隐层的权值：w_{ih}
- 中间隐层到输出层的权值：w_{ha}
- 传输函数为 f，其导数为 f'

在以上基本信息的基础上，可以得到每一个输出的误差为 t_i-ao_i。

所以可定义神经网络均方误差为：

$$e=\frac{1}{2}\sum_{i=1}^{i=q}(t_i-ao_i)^2$$

神经网络学习的最终目标就可以量化为上面的均方误差，也就是使得神经网络均方误差最小。该公式表示，神经网络误差 e 为每一个期望输出与实际输出的差的平方和。

根据最快下降法，要修正的权值就是 e 对 w 的偏导数：

$$\Delta w=-\eta\frac{\partial e}{\partial w}$$

其中 η 为学习步长。对最快下降法不了解的读者，可以查阅相关文档。这里只简要说明如下：所谓最快下降，就是使函数沿着某点的梯度方向前进，这是使函数值变化最快的方向。梯度就是函数对每一个变量的偏导数所构成的向量。因此，这里就有了 e 对 w 的

偏导数。

中间层到输出层权值变化根据该公式可以求得：

$$\Delta w_{ha} = -\eta \frac{\partial e}{\partial w_{ha}} = -\eta \sigma_a ho_h$$

上式中，w_{ha} 为隐层到输出层的权值，σ_a 为输出层第 a 个神经元输入对误差的敏感性，下标 a 取值为 1 到输出层神经元个数 q。σ_a 的定义为：

$$\sigma_a = \frac{\partial e}{\partial ai_a} = -(t_a - ao_a) f'(ai_a)$$

其中，ai_a 表示输出层第 a 个输出神经元的输入。σ_a 敏感性越大，表示节点的净输入变化对网络误差的影响也越大。

上式表明，中间隐层到输出层的权值调整，为神经元的误差、隐层神经元的输出以及传输函数在输出层净输入导数的三者的乘积。

输入层到中间隐层的权值变化公式更为复杂，这里不给出推导具体过程，直接给出结果：

$$\Delta w_{ih} = -\eta \frac{\partial e}{\partial w_{ia}} = -\eta \frac{\partial e}{\partial hi_h} \frac{\partial hi_h}{\partial w_{ih}} = \eta \sigma_h x_i$$

$$\sigma_h = \frac{\partial e}{\partial hi_h} = \left(\sum_{i=1}^{i=q} \sigma_i w_{hi} \right) f'(hi_h)$$

上面公式中 σ_h 为第 h 个中间隐层神经元的对误差的敏感性，σ_i 为第 i 个输出层神经元对误差的敏感性，w_{hi} 表示隐层到输出层神经元各向权值，i 取值为 1 到输出层神经元个数 q，w_{ih} 表示输入层到隐层的权值。可以看到 σ_h 是由它的后一层即输出层的敏感性反向推导得出的，这也正是反向传播命名的由来。σ_h 推导定义为后一层神经元（输出层）的误差敏感性的各向加权求和与传输函数在隐层净输入值导数的乘积。

至此，两个非常重要的权值更新公式可以给出：

$$\Delta w_{ha} = \eta (t_a - ao_a) f'(ai_a) ho_h$$

$$\Delta w_{ih} = \eta \left(\sum_{i=1}^{i=q} \sigma_i w_{hi} \right) f'(hi_h) x_i$$

这两个公式看似复杂，但实际上蕴含的道理并不复杂。当神经网络输出和期望输出有冲突时，如何调整各向权值呢？首先，对于某连接的敏感性越高，就应该对其权值进行越

大的调整，两者成正比，因为调整一个敏感性较低的连接意义不大，极端情况下，如果敏感性为 0，调整权值也没有任何意义。其次，与节点上一层的输出也成正比关系，因为上一层节点输出值越大，对结果的影响也越大，这种关系从后往前推导，最终在输入层将会使用原始的网络输入数据。

6.2.2 BP 学习算法与 BP 神经网络的实现

在了解 BP 学习算法的基本原理后，就可以动手实现它了。在本节中，将基于 Neuroph 神经网络框架和反向传播算法实现一个真正意义上的 BP 神经网络。

一个典型的 BP 神经网络的静态结构定义如下：

```
01 public class MlPerceptron extends NeuralNetwork<BackPropagation> {
02     .....
03     public MlPerceptron(List<Integer>neuronsInLayers, NeuronPropertiesneuron
       Properties) {
04         this.createNetwork(neuronsInLayers, neuronProperties);
05     }
06
07     protected void createNetwork(List<Integer>neuronsInLayers, NeuronPropertiesneuron
       Properties) {
08
09         // 设置神经网络类型
10         this.setNetworkType(NeuralNetworkType.MULTI_LAYER_PERCEPTRON);
11
12         // 输入神经元
13         NeuronPropertiesinputNeuronProperties = new NeuronProperties (InputNeuron.
           class, Linear.class);
14         Layer layer = LayerFactory.createLayer(neuronsInLayers.get(0), inputNeuron
           Properties);
15         layer.addNeuron(new BiasNeuron());
16         this.addLayer(layer);
17         // 创建隐层
18         Layer prevLayer = layer;
19         for (intlayerIdx = 1; layerIdx<neuronsInLayers.size(); layerIdx++) {
20             Integer neuronsNum = neuronsInLayers.get(layerIdx);
21             layer = LayerFactory.createLayer(neuronsNum, neuronProperties);
22             this.addLayer(layer);
23             // 最后一层不需要偏置
24             if(layerIdx != neuronsInLayers.size()-1)
25                 layer.addNeuron(new BiasNeuron());
26             // 连接各层
```

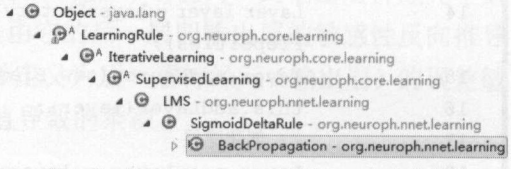
```
27                  if (prevLayer != null) {
28                      ConnectionFactory.fullConnect(prevLayer, layer);
29                  }
30
31                  prevLayer = layer;
32              }
33              NeuralNetworkFactory.setDefaultIO(this);
34              // 设置学习算法，此处为反向传播
35              this.setLearningRule(new BackPropagation());
36              // 初始化神经元的连接权值
37              this.randomizeWeights(new NguyenWidrowRandomizer(-0.7, 0.7));
38          }
39  }
```

相信读者对以上代码并不陌生，因为在前几章中已经出现过类似的代码片段，但在这里，依然有几个值得大家注意的地方。

从代码框架上看，这里使用 MlPerceptron 类来实现 BP 神经网络，其核心方法为 createNetwork()方法，它用于构建完整的网络系统。该方法接受两个参数，第一个参数 neuronsInLayers 指定各个层次上的神经元数量，第二个参数 neuronProperties 指定网络中的神经元属性，比如神经元类型、传输函数、输入函数等。

从构造流程上看，createNetwork()方法首先构造了输入层，并设置偏置神经元模拟中间层的偏置（代码第 15 行），接着根据指定的神经元数量创建隐层和输出层，最后在第 35 行设置了该神经网络的学习算法为反向传播，在第 37 行使用 Nguyen-Widrow 方法初始化网络权值。

接着，重点关注 BackPropagation 学习算法。Neuroph 有良好的系统结构，各个模块之间分工明确，在 BackPropagation 类中，并不包含反向传播算法的所有内容。所以有必要先回顾一下 Neuroph 框架中的学习算法的建模方式（如图 6-6 所示）。



▲图 6-6　BackPropagation 在 Neuroph 中的位置

BackPropagation 继承自 LearningRule，LearningRule 是所有学习算法的父类。它包含一个重要的抽象方法：

```
abstract public void learn(DataSettrainingSet)
```

learn()方法接受一个训练数据集，并对网络进行训练。BackPropagation 算法属于有监督的迭代训练，因此从继承结构上看，它隶属于 IterativeLearning 和 SupervisedLearning

算法。这里的迭代指网络根据给定的训练数据在有限的循环周期内不断调整权值，直到误差到可接受范围或者达到最大的迭代次数。有监督的学习方式表示 BackPropagation 需要使用期望值来调衡自身权值，也就是每次网络的输出都可以找到对应的误差，存在一个可靠的指标指导网络的训练。IterativeLearning 提供的核心逻辑如下：

```
final public void learn(DataSettrainingSet) {
      setTrainingSet(trainingSet); // set this field here su subclasses can access
it
      onStart();

      while (!isStopped()) {
            beforeEpoch();
            //训练网络
            doLearningEpoch(trainingSet);
            this.currentIteration++;
            afterEpoch();

            // 是否达到停止的条件，如误差在可接受范围内，
            // 或者达到最大迭代次数
            if (hasReachedStopCondition()) {
                  stopLearning();
            }
      ...
}
```

可以看到 IterativeLearning 实现了 learn()方法，并进行迭代学习。如果达到停止条件则停止学习，否则学习过程将一直持续进行。其中重要的 doLearningEpoch()方法为该类的抽象方法，它将在子类中实现，并负责具体的学习过程。SupervisedLearning 类就提供了一个有监督的学习实现：

```
public void doLearningEpoch(DataSettrainingSet) {

      // 对于所有的训练数据，需要依次进行学习
      Iterator<DataSetRow> iterator = trainingSet.iterator();
      while (iterator.hasNext() && !isStopped()) {
            DataSetRowdataSetRow = iterator.next();
            // 学习其中一条数据
            this.learnPattern(dataSetRow);
      }

      // 得到均方误差
```

```
        this.totalNetworkError = errorFunction.getTotalError();

        // 判断是否达到停止条件，比如误差已经在可接受范围内
        if (hasReachedStopCondition()) {
            stopLearning();
        }
    }
```

其中 leanPattern()用于学习一条训练数据，它在 SupervisedLearning 中的实现如下：

```
protected void learnPattern(DataSetRowtrainingElement) {
    double[] input = trainingElement.getInput();
    this.neuralNetwork.setInput(input);
    //根据输入计算神经网络的输出
    this.neuralNetwork.calculate();
    double[] output = this.neuralNetwork.getOutput();
    double[] desiredOutput = trainingElement.getDesiredOutput();
    //根据网络的实际输出和期望计算均方差
    double[] outputError = this.calculateOutputError(desiredOutput, output);
    // 误差累计
    errorFunction.addOutputError(outputError);
    // 根据每个输出的误差更新网络的各向权值
    this.updateNetworkWeights(outputError);
}
```

可以看到，在单条训练数据的学习中，首先根据输入计算神经网络的实际输出，根据期望值与实际输出计算误差，并将其累加到均方差中。

以上代码中的 errorFunction 就是均方误差函数。网络的均方误差定义为：

$$e=\frac{1}{2}\sum_{i=1}^{i=q}(t_i-ao_i)^2$$

网络学习的根本就是将均方误差减少到最低，因此网络学习的终止条件通常可设定为均方误差在可接受范围内。在实际实现中，均方误差被定义为所有训练数据均方误差的平均值（全局误差），即：

$$e=\frac{1}{2m}\sum_{k=1}^{m}\sum_{i=1}^{i=q}(t_i-ao_i)^2$$

其实现如下：

```
public double getTotalError() {
    // 获取均方误差时，使用平均值
```

```
        returntotalSquaredErrorSum/n;
    }

    public void addOutputError(double[] outputError) {
        doubleoutputErrorSqrSum = 0;
        // 计算一条训练数据的均方误差
        for (double error : outputError) {
            outputErrorSqrSum += (error * error) * 0.5;
        }
        // 多条训练数据均方误差累加
        this.totalSquaredErrorSum += outputErrorSqrSum;
    }
```

在每一条数据的学习中，网络都要根据每个神经元的误差来调整网络的各向权值。在SupervisedLearning 类中，使用 updateNetworkWeights()方法来实现这一功能（参考learnPattern()函数），该方法接受一次训练的各个输出层神经元的误差，并以此为基础调整网络的各向权值。updateNetworkWeights()在 SupervisedLearning 类中为抽象函数，因为监督学习只负责给出基本的误差计算和比对，达到期望输出对网络的指导效果，而更进一步的权值调整将在其子类中实现，不同的学习方法有着不同的权值调整策略。BP学习算法在 BackPropagation 类中的实现如下：

```
protected void updateNetworkWeights(double[] outputError) {
    this.calculateErrorAndUpdateOutputNeurons(outputError);
    this.calculateErrorAndUpdateHiddenNeurons();
}
```

权值调整分为两个阶段，首先会调整输出层的权值：

```
protected void calculateErrorAndUpdateOutputNeurons(double[] outputError) {
    int i = 0;
    for(Neuron neuron : neuralNetwork.getOutputNeurons()) {
        //如果没有误差，则设置神经元误差为 0
        if (outputError[i] == 0) {
            neuron.setError(0);
            i++;
            continue;
        }

        //有误差则再计算误差，取得传输函数
        TransferFunctiontransferFunction = neuron.getTransferFunction();
        doubleneuronInput = neuron.getNetInput();
        //计算传输函数在输入点的导数和神经元误差的乘积σa=(ta-aoa)f'(aia)
```

```
            //即该神经元的敏感性
            double delta = outputError[i] * transferFunction.getDerivative(neuronInput);
            //设置神经元的敏感性
            neuron.setError(delta);

            //更新该神经元的权重
            this.updateNeuronWeights(neuron);
            i++;
        } // for
    }
```

为保证阅读的完整性，这里再次给出 updateNeuronWeights()方法的实现：

```
protected void updateNeuronWeights(Neuron neuron) {
    // 取得神经元误差
    doubleneuronError = neuron.getError();

    // 根据所有的神经元输入迭代学习
    for (Connection connection : neuron.getInputConnections()) {
        // 神经元的一个输入
        double input = connection.getInput();
        // 计算权值的变更Δwh_a=ησ_a ho_h
        doubleweightChange = this.learningRate * neuronError * input;
        // 更新权值
        Weight weight = connection.getWeight();
        weight.weightChange = weightChange;
        weight.value += weightChange;
    }
}
```

从这两段代码实现了上一章的权值更新公式：

$$\Delta w_{ha} = \eta \sigma_a ho_h$$

$$\sigma_a = -(t_a - ao_a) f'(ai_a)$$

在完成输出层的权值调整后，进行隐层的权值调整。隐层权值调整与输出层最大的不同在于对神经元敏感性的计算：

$$\Delta w_{ih} = \eta \sigma_h x_i$$

$$\sigma_h = \left(\sum_{i=1}^{i=q} \sigma_i w_{hi} \right) f'(hi_h)$$

隐层的权值更新代码如下：

```
protected void calculateErrorAndUpdateHiddenNeurons() {
        Layer[] layers = neuralNetwork.getLayers();
    for (intlayerIdx = layers.length - 2; layerIdx> 0; layerIdx--) {
          for(Neuron neuron : layers[layerIdx].getNeurons()) {
              //计算隐层敏感度
              doubleneuronError = this.calculateHiddenNeuronError(neuron);
              neuron.setError(neuronError);
              this.updateNeuronWeights(neuron);
          }
    } /
}
```

其中，calculateHiddenNeuronError()方法计算了隐层的敏感度：

```
protected double calculateHiddenNeuronError(Neuron neuron) {
    doubledeltaSum = 0d;
    for(Connection connection : neuron.getOutConnections()) {
        //根据后一层神经元的敏感性计算当前层神经元的敏感性
        //这里计算了 ∑_{i=1}^{i=q} σ_i w_hi
        double delta = connection.getToNeuron().getError()
                * connection.getWeight().value;
        deltaSum += delta;
    }
    //计算敏感性 (∑_{i=1}^{i=q} σ_i w_hi) f'(hi_h)
    TransferFunctiontransferFunction = neuron.getTransferFunction();
    doublenetInput = neuron.getNetInput();
    double f1 = transferFunction.getDerivative(netInput);
    doubleneuronError = f1 * deltaSum;
    returnneuronError;
}
```

从上述代码可以看到，隐层的敏感性由后一层神经元的敏感性进行加权求和反向传播得到。至此，整个 BP 神经网络学习算法的主要流程已经介绍完毕，其基本步骤可以概括为：

- 读取训练数据；
- 计算各输出神经元误差；
- 计算神经网络均方差；
- 计算输出层敏感度；

- 调整输出层权值；
- 根据输出层敏感度和权值反向计算中间隐层的敏感度；
- 根据隐层的敏感度调整隐层权值。

重复以上步骤，直到误差可接受或达到最大迭代次数。

6.3 BP 神经网络细节优化

在前文中，已经介绍了 BP 神经网络的学习算法，并根据该算法给出了一个实现的基本流程。虽然这个基本流程已经足够实现一个 BP 神经网络，但仍然有一些实现的细节，需要在本节中做进一步的说明和探讨。

6.3.1 随机化权值的方式

神经网络的学习算法本质上就是对神经元权值的调整。神经元的权值会影响神经网络的学习效果。在前两节中已经提到，神经网络的初始化权值是一些随机数，网络根据训练样本的情况不断调整权值，直到达到可接受的状态。因此，可以想象到，如果网络拥有一个合理的权值，那么其学习时间必然会缩短，反之，它的学习时间就会变长。

一个最简单的初始化方法就是使用范围随机化，也就是指定一个随机数的边界，使得所有的网络权值都落在一个合理的范围内。相对于杂乱无章的权值，这个方法对网络的可训练性是有所改善的。通常，取值在−1～1 都是可行的。在 Neuroph 框架中，就有一个范围随机化的实现：

```
public class RangeRandomizer extends WeightsRandomizer {
    ......省略部分代码
    protected double nextRandomWeight() {
        return min + randomGenerator.nextDouble() * (max - min);
    }
}
```

另一个更有效的改动，是使用 Nguyen-Widrow 随机化算法。Nguyen-Widrow 是最有效的神经网络权值初始化算法之一。它是由 Derrick Nguyen 和 Bernard Widrow 共同发明的，故因此命名。Nguyen-Widrow 算法是对范围随机算法的一种改进，它首先使用范围算法，将初始化权值都限定在一个范围内。紧接着计算一个 β 值：

$$\beta = 0.7h^{\frac{1}{i}}$$

其中，h 是中间隐层神经元数量，i 为输入层神经元数量。

然后，为每一层计算欧几里得范数，也就是向量的距离：

$$n = \sqrt{\sum_{i=0}^{i<w_{max}} w_i^2}$$

最后，根据 β 和 n 可以进行权值的调整：

$$w_{\tau+1} = \frac{\beta w_{\tau+1}}{n}$$

Nguyen-Widrow 算法是本书使用的默认算法，它基于范围随机算法实现。它的完整实现如下，读者可以参考上述算法步骤：

```
public class NguyenWidrowRandomizer extends RangeRandomizer {

    publicNguyenWidrowRandomizer(double min, double max) {
    super(min, max);
    }

    @Override
    public void randomize(NeuralNetworkneuralNetwork) {
        super.randomize(neuralNetwork);

        intinputNeuronsCount = neuralNetwork.getInputNeurons().length;
        inthiddenNeuronsCount = 0;

        for (int i = 1; i <neuralNetwork.getLayersCount() - 1; i++) {
            hiddenNeuronsCount += neuralNetwork.getLayerAt(i).getNeuronsCount();
        }

        //根据隐层和输入层数量计算β
        double beta = 0.7 * Math.pow(hiddenNeuronsCount, 1.0 / inputNeuronsCount);

        for (Layer layer : neuralNetwork.getLayers()) {
            //对于每一层，分别计算范数
            double norm = 0.0;
            for (Neuron neuron : layer.getNeurons()) {
            for (Connection connection : neuron.getInputConnections()) {
                double weight = connection.getWeight().getValue();
                norm += weight * weight;
```

```
                }
            }
            norm = Math.sqrt(norm);

            //调整各向权值：β* 旧权值 / norm
            for (Neuron neuron : layer.getNeurons()) {
                for (Connection connection : neuron.getInputConnections()) {
                    double weight = connection.getWeight().getValue();
                    weight = beta * weight / norm;
                    connection.getWeight().setValue(weight);
                }
            }
        }
    }
}
```

6.3.2　Sigmoid 函数导数的探讨

另外一个有意思的话题是有关 Sigmoid 函数的导数实现。前面已经提到，权值的调整和传输函数的导数相关。回忆两个敏感性公式，都需要计算传输函数的导数：

$$\sigma_a = -(t_a - ao_a) f'(ai_a)$$

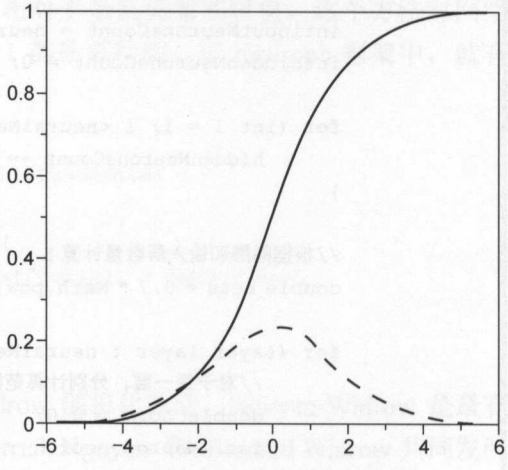
$$\sigma_h = \left(\sum_{i=1}^{i=q} \sigma_i w_{hi} \right) f'(hi_h)$$

Sigmoid 函数是最常用的传输函数，根据定义其导数如下。

Sigmoid 函数：$a=1/(1+exp(-n))$

Sigmoid 导数：$a'=a*(1-a)$

两者的图形如图 6-7 所示。



▲图 6-7　Sigmoid 函数及其导数

其中实线表示 Sigmoid 函数，虚线则是对应的导数。可以看到，在自变量很大或者很小时，其导数值会非常接近于 0。于是就产生了一个叫作扁平的点的问题（Flat Spot Problem）。在这种情况下，得到的Δw 也接近于 0。于是，每次迭代学习的效果就非常差，权值几乎得不到改进，从而加长了学习时间。因此，使用以下公式来解决该问题：

$$a(1-a)+0.1$$

以此得到的求 Sigmoid 导数的具体实现如下：

```
//求 Sigmoid 导数值
public double getDerivative(double net) {
    double derivative = this.slope * this.output * (1d - this.output) + 0.1;
    return derivative;
}
```

6.4 带着算法重回异或问题

现在我们已经有了完整的 BP 神经网络的实现，作为第一个 BP 神经网络的案例，就让我们再次回顾一下异或问题吧！与 6.1.3 节不同的是，这次，神经网络将会通过自学习而非人工构造来掌握异或算法。

在异或问题中，需要一个二值输出（神经元的输出只有 0 和 1），因此定义一个二值输出的多层感知机 MlPerceptronBinOutput，其核心构造方法如下：

```
1 protected void createNetwork(List<Integer>neuronsInLayers, NeuronPropertiesneuron
Properties) {
2       this.setNetworkType(NeuralNetworkType.MULTI_LAYER_PERCEPTRON);
3       // 输入层
4       NeuronPropertiesinputNeuronProperties = new NeuronProperties(InputNeuron.class,
        Linear.class);
5       Layer layer = LayerFactory.createLayer(neuronsInLayers.get(0), inputNeuron
        Properties);
6       layer.addNeuron(new BiasNeuron());
7       this.addLayer(layer);
8
9       Layer prevLayer = layer;
10       intlayerIdx = 1;
11       for (layerIdx = 1; layerIdx<neuronsInLayers.size()-1; layerIdx++) {
12             Integer neuronsNum = neuronsInLayers.get(layerIdx);
13             // 中间层，传输函数为 Sigmoid
14             layer = LayerFactory.createLayer(neuronsNum, neuronProperties);
15             layer.addNeuron(new BiasNeuron());
16             this.addLayer(layer);
17             if (prevLayer != null) {
18                   ConnectionFactory.fullConnect(prevLayer, layer);
19             }
```

```
20              prevLayer = layer;
21          }
22
23          Integer neuronsNum = neuronsInLayers.get(layerIdx);
24          NeuronPropertiesoutProperties=new NeuronProperties();
25          // 输出层为 Step 的二值函数
26          outProperties.put("transferFunction", org.neuroph.core.transfer.Step.class);
27          layer = LayerFactory.createLayer(neuronsNum, outProperties);
28          this.addLayer(layer);
29          ConnectionFactory.fullConnect(prevLayer, layer);
30          prevLayer = layer;
31
32          NeuralNetworkFactory.setDefaultIO(this);
33
34          // 学习算法为反向传播
35          this.setLearningRule(new BackPropagation());
36          // Nguyen-Widrow 方法初始化权值
37          this.randomizeWeights(new NguyenWidrowRandomizer(-0.7, 0.7));
38  }
```

上述代码构造了一个拥有二值输出的神经网络。在第 14 行，创建了神经网络的中间隐层，并使用 Sigmoid 作为神经元的传输函数。第 26 行，设置 Step 函数作为输出层传输函数，实现了神经网络输出的二值化。在第 35 行，设置网络的学习算法为反向传播。在第 37 行，使用了 Nguyen-Widrow 方法初始化各向权值。

在以上代码的基础上，可以使用 MlPerceptronBinOutput 网络记忆异或问题：

```
1 // 训练数据
2 DataSettrainingSet = new DataSet(2,1);
3 trainingSet.addRow(new DataSetRow(new double[]{0, 0}, new double[]{0}));
4 trainingSet.addRow(new DataSetRow(new double[]{0, 1}, new double[]{1}));
5 trainingSet.addRow(new DataSetRow(new double[]{1, 0}, new double[]{1}));
6 trainingSet.addRow(new DataSetRow(new double[]{1, 1}, new double[]{0}));
7 // 2 个输入数据，4 个神经元隐层，1 个输出
8 MlPerceptronmyPerceptron = new
MlPerceptronBinOutput(TransferFunctionType.SIGMOID,2,4,1);
9 LearningRulelearningRule = myPerceptron.getLearningRule();
10 learningRule.addListener(this);
11 // 训练神经网络
12 System.out.println("Training neural network...");
13 myPerceptron.learn(trainingSet);
```

```
14 // 测试神经网络
15 System.out.println("Testing trained neural network");
16 testNeuralNetwork(myPerceptron, trainingSet);
```

以上代码使用 MlPerceptronBinOutput 记忆了异或问题。首先在第 2～6 行，构造了异或问题的训练数据。在第 8 行，使用 MlPerceptronBinOutput 构造了一个具体的神经网络，它拥有 2 个输入、4 个神经元的中间隐层和 1 个输出（只包含 0 和 1 的二值输出）。在第 13 行，使用训练数据训练该神经网络。最后，在第 16 行，测试神经网络的正确性。运行以上代码可以得到类似以下的输出：

```
Training neural network...
============
classorg.neuroph.core.events.LearningEvent
iterate:1
0.15003553684472318
.....省略部分权值信息.....
classorg.neuroph.core.events.LearningStoppedEvent
iterate:314
...省略部分权值信息...
0.3500355368447232
Testing trained neural network
Input: [0.0, 0.0] Output: [0.0]
Input: [0.0, 1.0] Output: [1.0]
Input: [1.0, 0.0] Output: [1.0]
Input: [1.0, 1.0] Output: [0.0]
```

从该输出中可以看到，第一次迭代学习中网络的权值信息，以及最后一次迭代中网络的权值信息。此网络共进行了 314 次迭代后，达到可接受的误差。最后，通过原始的训练数据测试神经网络，可以看到网络对异或的四种情况均给出了正确的输出。

6.5 总结

本章详细介绍了 BP 神经网络的原理与具体实现。作为最重要也是使用最为广泛的神经网络之一，BP 神经网络在整个神经网络发展历史中，有着极其重要的作用。除了对理论和基本应用的介绍外，本章还从工程实践角度，介绍了实现 BP 网络的一些可用的优化方法和注意事项。

第 7 章　BP 神经网络的案例

第 6 章详细介绍了 BP 神经网络的基本原理和实现。那么，BP 神经网络究竟能带给我们什么呢？在本章中，我们将枚举一些 BP 神经网络的典型应用，比如分类问题、拟合问题、语音识别，甚至是手写体识别问题。通过本章的学习，读者应该对 BP 神经网络的应用有更深刻的认识。

7.1 奇偶性判别问题

7.1.1　问题描述

奇偶性判别问题很简单，即给出任意非零整数，要求系统输出这个数字的正负和奇偶特性。比如，输入数字 8，应该能够输出正整数；输入数字–99，应该输出负奇数。虽然这个问题可以很容易地使用数学方法求解，计算机也能够很容易地使用数学公式判断一个数字的奇偶性和正负性，但是，这个实例的重点在于：在不告知神经网络奇偶性或正负性判别公式的前提下，要求神经网络自己发现这个规律。使用这个方法，即使在将来遇到人类尚未识别的某些规律，一样可以使用神经网络作为辅助进行数据分析。

7.1.2　代码实现

为了使用神经网络来解决这个问题，首先必须选择一种编码方式，即决定输入的数字以何种方式传递给神经网络。由于整数通常使用 32 位表示，所以在此使用 32 个神经元分别表示整数的每一个比特位。以下代码显示了如何将一个整数转为一个 32 位的输入，该输入将直接传递给神经网络（有兴趣的读者可以参考随书代码中的 ParityCheck 类）：

```
public static double[] int2double(int i){
    double[] re=new double[32];
    for(int j=0;j<32;j++){
        re[j]=(double)((i>>j)&1);
    }
    return re;
}
```

由于神经网络有 4 种输出的可能性，因此可以设定输出神经元也为四个，分别表示正偶数、负偶数、正奇数、负奇数四种情况。在这里，进行如下定义。

输出：0001　正偶数

输出：0010　负偶数

输出：0100　正奇数

输出：1000　负奇数

根据以上对输入和输出的定义，构造一个拥有 32 个输入、10 个中间隐层、4 个输出的多层感知机神经网络，并使用 Sigmoid 作为其传输函数：

```
MlPerceptronmyMlPerceptron = new MlPerceptron(TransferFunctionType.SIGMOID, 32, 10, 4);
```

为了提高神经网络的判断准确性，可以设置一个小一些的期望误差：

```
LearningRulelearningRule = myMlPerceptron.getLearningRule();
learningRule.addListener(this);
//设置误差
((SupervisedLearning)learningRule).setMaxError(0.0001d);
```

接着，需要对该网络进行训练。首先需要求得正确解，对于任意输入整数，给出一个正确的输出：

```
public static double[] int2prop(int i){
    //正偶数
    double[] pe={0d,0d,0d,1d};
    //负偶数
    double[] ne={0d,0d,1d,0d};
    //正奇数
    double[] po={0d,1d,0d,0d};
    //负奇数
    double[] no={1d,0d,0d,0d};
```

```
        if(i>0 && i%2==0){
            returnpe;
        }else if(i<0 && i%2==0){
            return ne;
        }else if(i>0 && i%2!=0){
            returnpo;
        }else if(i<0 && i%2!=0){
            return no;
        }
        returnpe;
}
```

训练 2000 个整数：

```
DataSettrainingSet = new DataSet(32, 4);
//对 2000 条数据进行训练
for(int i=0;i<2000;i++){
      int in=new Random().nextInt();
      trainingSet.addRow(new DataSetRow(int2double(in), int2prop(in)));
}
….
myMlPerceptron.learn(trainingSet);
```

由于这里构建的网络并不会给出一个明确的 0 或者 1 的输出，它给出的输出信号为浮点数，因此需要制定一套规则将其转为预定的 0 和 1。这里使用最简单也是最有效的方法，将活跃度高的神经元输出设置为 1，其他均为 0。比如，若神经网络最终输出为 0.9、0.8、0.7、0.6，将被转换为 1、0、0、0，其实现如下：

```
public class Utils {
    public static double[] competition(double[] d){
        double[] output=d;
        double[] re=new double[output.length];
        intmaxIndex=0;
        doublemaxValue=Double.MIN_VALUE;
        for(int i=0;i<output.length;i++){
            if(output[i] >maxValue){
                maxIndex=i;
                maxValue=output[i];
            }
        }
        for(int i=0;i<re.length;i++){
            if(i==maxIndex){
                re[i]=1;
            }else{
```

```
                re[i]=0;
            }
        }
        return re;
    }
}
```

经过以上训练后，该神经网络便已经记忆了合适的权值，并可以对未知的数据进行正确的分类。这里，任意取 50000 个整数，让神经网络判断这 50000 个整数的类别。

```
public static void testNeuralNetwork(NeuralNetworkneuralNet) {
    intbadcount=0;
    int COUNT=50000;
    //判断 50000 个整数的类别
    for(int i=0;i<COUNT;i++){
        int in=new Random().nextInt();
        double[] inputnumber=int2double(in);
        neuralNet.setInput(inputnumber);
        neuralNet.calculate();
        double[] networkOutput = neuralNet.getOutput();
        //转为只包含 0 和 1 的输出，取最活跃的神经网络为 1
        networkOutput=Utils.competition(networkOutput);
        //神经网络给出的分类
        String networkOutputDisplay=networkOutputDisplay(networkOutput);
        //该数字正确的分类
        String cc=correctClassify(in);
        System.out.print(in+" "+networkOutputDisplay+" ");
        if(i%50==0){
        System.out.println();
        }
        //如果神经网络分类错误，则记录错误数
        if(!cc.equals(networkOutputDisplay)){
        badcount++;
        System.out.print("判别错误:"+in);
        System.out.print(" correctClassify="+cc);
        System.out.println(" networkOutputDisplay="+networkOutputDisplay);
        }
    }
    //给出网络分类的正确率
    System.out.println();
    System.out.println("正确率: "+(COUNT-badcount*1.0)/COUNT*100.0+"%");
}
```

以上代码随机生成了 50000 个整数，使用神经网络对每一个整数进行分类，同时求得该整数正确的分类，如果神经网络判断错误，则记录错误数，并最终输出正确率。在笔者

的计算机上，该程序的一个可能输出如下：

```
Training neural network...
1. iteration : 0.2941280576252788
2. iteration : 0.038846640862080145
3. iteration : 0.008070477461699761
.....
17. iteration : 1.3440824538874392E-4
18. iteration : 1.1938161177508181E-4
19. iteration : 1.0673838109503141E-4
20. iteration : 9.59991031399463E-5
20. iteration : 9.59991031399463E-5
Testing trained neural network
29693272 正偶数
-493350427 负奇数 362184868 正偶数  377792354 正偶数 -88134227 负奇数 51717429 正奇数
-780282111 负奇数
......
正确率: 100.0%
```

从程序的输出可以看到，神经网络进行了 20 次迭代学习后达到误差的接受范围。学习过程中，误差逐步减小，训练完成后，对 50000 个未知数字进行分类，正确率达到 100%。

7.2 函数逼近

7.2.1 问题描述

函数逼近问题指的是，在不给出函数关系式的前提下，仅通过大量的函数值对应实例对神经网络进行训练，使得神经网络可以根据一个未知的自变量预测对应的应变量值。理论上说，函数逼近和数值预测的原理是相同的，数值预测可以看作在函数关系不明确状态下的函数关系的逼近。

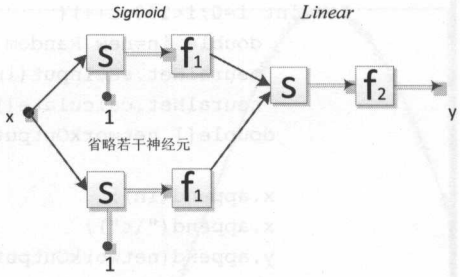
7.2.2 代码实现

本节将使用 BP 神经网络，模拟逼近一正弦函数：

$$y = 1 + \sin\left(\frac{i\pi}{4}\right)x$$

该函数的振荡频率与参数 i 有关，i 越大则振荡周期越短，频率越高，函数的变化率也越大。为了使用神经网络逼近该函数，这里采用 *Sigmoid+Linear* 复合作为传输函数的

结构，如图 7-1 所示。



▲图 7-1 网络逼近函数

该网络接受一个输入 x，也就是函数的自变量，输出函数值 y。总体上说分为两层，第一层为 *Sigmoid* 传输层，第二层为 *Linear* 传输层。其中，*Sigmoid* 层的神经元个数根据实验情况可以调整，这里默认取 4 个神经元。

以下代码构造了上述结构的神经网络，拥有 1 个输入神经元、4 个隐层 *Sigmoid* 神经元和 1 个输出层 *Linear* 神经元（有兴趣的读者可以参考随书代码的 FunctionAppro 类）：

```
MlPerceptron myMlPerceptron = new MlPerceptronLineOutput(1,4,1);
myMlPerceptron.setLearningRule(new BackPropagation());
myMlPerceptron.getLearningRule().setMaxError(maxError);
```

在区间[−2,2]中，随机取 2000 个数字作为训练样本，提供网络进行训练：

```
DataSet trainingSet = new DataSet(1, 1);
 for(int i=0;i<2000;i++){
      double in=new Random().nextDouble()*4-2;
      double out=func(in);
      trainingSet.addRow(new DataSetRow(new double[]{in}, new double[]{out}));
 }
 LearningRule learningRule = myMlPerceptron.getLearningRule();
 learningRule.addListener(this);
 System.out.println("Training neural network...");
 myMlPerceptron.learn(trainingSet);
```

其中 func()函数就是目标函数 $y=1+\sin\left(\frac{i\pi}{4}\right)x$：

```
public double func(double x){
    return 1+Math.sin(Math.PI*i/4*x);
}
```

训练完成后，在[−2,2]区间内任取 100 个数字进行预测，并将测试用的数字和预测值记录到指定文件中，方便后续绘图：

```
public static void testNeuralNetwork(NeuralNetwork neuralNet) throws IOException {
     StringBuffer x=new StringBuffer();
     StringBuffer y=new StringBuffer();
```

```
        for(int i=0;i<100;i++){
                double in=new Random().nextDouble()*4-2;
                neuralNet.setInput(in);
                neuralNet.calculate();
               double[] networkOutput = neuralNet.getOutput();

               x.append(in);
               x.append("\t");
               y.append(networkOutput[0]);
               y.append("\t");
        }
        //输出到文件使用 scilab 绘制图像
        FileUtils.writeStringToFile(new File("C:/x.txt"), x.toString());
        FileUtils.writeStringToFile(new File("C:/y.txt"), y.toString());
}
```

取 *i*=2,maxError=0.01，执行以上代码，得到输出：

```
Training neural network...
1. iteration : 0.04187563481078222
2. iteration : 0.02138074249781963
......
8. iteration : 0.032759170965009
9. iteration : 0.013395231603537337
10. iteration : 0.004380492244211028
```

可以看到，在 10 次迭代学习后，网络达到误差范围内，并在 C 盘下形成 x.txt 和 y.txt 两个文件，分别存储了 100 个随机数以及网络对它的函数逼近预测值，使用 scilab 绘制函数图像：

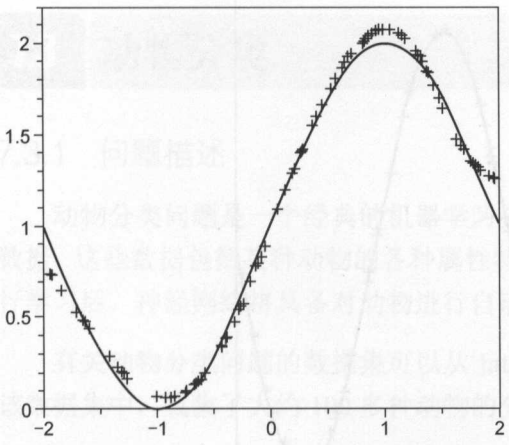
```
clf();
i=2;
t = read('C:/x.txt', 1, 100);
t1 = read('C:/y.txt', 1, 100);

x=[-2:0.04:2]
plot(t,t1,'k+',x,sin(%pi*i/4*x)+1,'k');
```

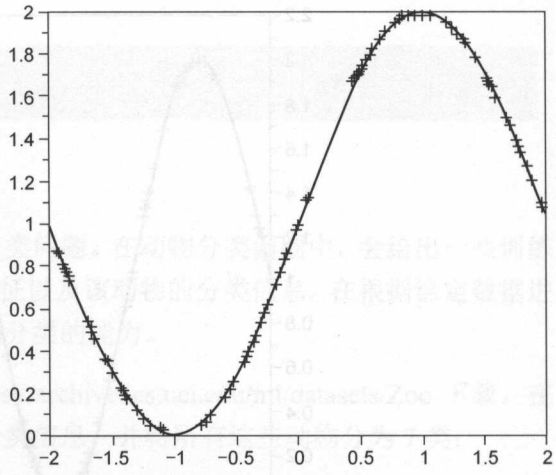
以上代码将网络的预测数据使用“+”号绘制，使用连续实线绘制目标函数，得到结果如图 7-2 所示。当然，如果读者愿意，也可以使用其他工具绘图，甚至可以在 Java 中直接绘图。

从图 7-2 中可以看到，预测结果基本落在期望值周围，并保持了相同的趋势。为了得到更精确的结果，可以减小期望误差。将 maxError 设置为 0.0001，经过 20 次左右的迭代学习，得到结果如图 7-3 所示。

很明显，在减少误差后，所有的预测点基本上与期望值完全吻合了。

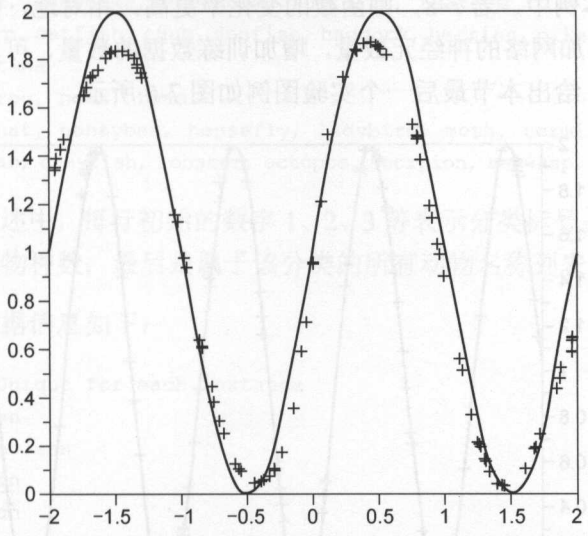


▲图 7-2 *i*=2,maxError=0.01,4 个 Sigmoid 神经元的预测结果



▲图 7-3 *i*=2,maxError=0.0001,4 个 Sigmoid 神经元的预测结果

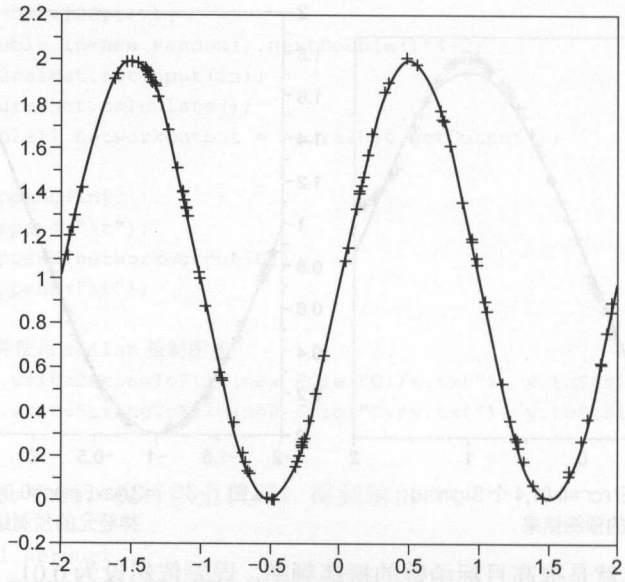
现在令 *i*=4，也就是提高目标函数的振荡频率，误差依然设为 0.01。让网络学习该函数后可以发现，该网络已经无法将目标函数的误差控制在期望值内。这说明，随着函数拐点增多，需要更多的神经元参与到网络中来。调整为 8 个神经元后，得到结果如图 7-4 所示。



▲图 7-4 *i*=4,maxError=0.01,8 个 Sigmoid 神经元的预测结果

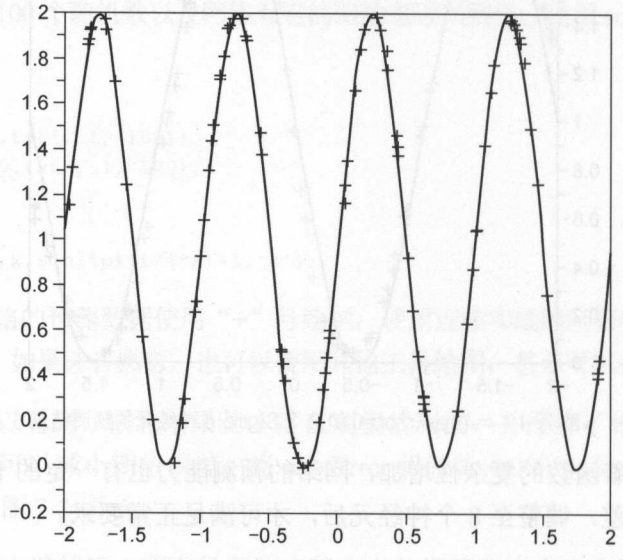
可以看到，随着函数的复杂性增加，网络的预测能力也有一定的下降，4 个神经元已经无法满足预期误差，调整至 8 个神经元后，才可满足正常要求。

同理，设置误差为 0.0001，可以得到更精确的预测如图 7-5 所示。



▲图 7-5　/=4,maxError=0.0001,8 个 Sigmoid 神经元的预测结果

理论上说，神经网络可以逼近任意连续函数，但当函数的变化曲率很大时，这种逼近会越来越困难。在本例中，若 i=8，则函数的变化率更高，相对地，网络的逼近能力就会显得很弱，此时，增加网络的神经元数量，增加训练数据的数量，可以在一定程度上提升网络的能力。最后，给出本节最后一个实验图例如图 7-6 所示。



▲图 7-6　/=8,maxError=0.0001,16 个 Sigmoid 神经元，20 万条训练数据的预测结果

7.3 动物分类

7.3.1 问题描述

动物分类问题是一个经典的机器学习分类问题。在动物分类问题中，会给出一些训练数据，这些数据包括某种动物的各种属性特征以及该动物的分类信息。在根据给定数据进行学习后，神经网络将具备对动物进行自动分类的能力。

有关动物分类问题的数据集可以从 https://archive.ics.uci.edu/ml/datasets/Zoo 下载。在该数据集中，收集了大约 100 多种动物的分类信息，并将所有这些动物分为 7 类：

```
1 -- (41) aardvark, antelope, bear, boar, buffalo, calf, cavy, cheetah, deer, dolphin,
elephant, fruitbat, giraffe, girl, goat, gorilla, hamster, hare, leopard, lion, lynx,
mink, mole, mongoose, opossum, oryx, platypus, polecat, pony, porpoise, puma, pussycat,
raccoon, reindeer, seal, sealion, squirrel, vampire, vole, wallaby,wolf
2 -- (20) chicken, crow, dove, duck, flamingo, gull, hawk, kiwi, lark, ostrich, parakeet,
penguin, pheasant, rhea, skimmer, skua, sparrow, swan, vulture, wren
3 -- (5) pitviper, seasnake, slowworm, tortoise, tuatara
4 -- (13) bass, carp, catfish, chub, dogfish, haddock, herring, pike, piranha, seahorse,
sole, stingray, tuna
5 -- (4) frog, frog, newt, toad
6 -- (8) flea, gnat, honeybee, housefly, ladybird, moth, termite, wasp
7 -- (10) clam, crab, crayfish, lobster, octopus, scorpion, seawasp, slug, starfish, worm
```

在上面的分类描述中，每行初始的数字 1、2、3 等表示分类标号；后续括号中的数字表示属于该分类的动物种数；最后是属于该分类的所有动物名称列表，并以逗号分隔。

给出的每一行数据信息如下：

```
1. animal name: Unique for each instance
2. hair:    Boolean
3. feathers:    Boolean
4. eggs:    Boolean
5. milk:    Boolean
6. airborne:    Boolean
7. aquatic:    Boolean
8. predator:    Boolean
9. toothed:    Boolean
10. backbone:  Boolean
11. breathes:  Boolean
```

```
12. venomous:  Boolean
13. fins:Boolean
14. legs:Numeric (set of values: {0,2,4,5,6,8})
15. tail:Boolean
16. domestic: Boolean
17. catsize:  Boolean
18. type:  Numeric (integer values in range [1,7])
```

一共 18 列。第 1 列为动物名称，第 2～17 列表示动物的特征，第 18 列为分类指导数据，即期望输出。第 2～17 列的信息中，包含了一些重要的生物学特征，比如第 2 列 hair 表示是否有头发，第 3 列 feathers 表示是否有羽毛，第 4 列 eggs 表示是否卵生。除了这些布尔类型的数据外，还有些特征为整数值，比如第 14 列的 legs 表示该动物有几条腿。

第 18 列的数值范围为 1～7，也正是表示分类的结果。因此动物分类问题的本质就是将数据的第 2～17 列作为神经网络的输入，预测或者说是输出第 18 列值。以下是部分训练数据：

```
frog,0,0,1,0,0,1,1,1,1,1,1,0,4,0,0,0,5
fruitbat,1,0,0,1,1,0,0,1,1,1,0,0,2,1,0,0,1
giraffe,1,0,0,1,0,0,0,1,1,1,0,0,4,1,0,1,1
girl,1,0,0,1,0,0,1,1,1,1,0,0,2,0,1,1,1
```

有兴趣的读者可以根据每列的含义，检查一下其数据是否有误。

7.3.2　问题分析

为了可以更好地求解并验证神经网络的效果，在本次试验中，将原始数据分为两组。一组用于测试，另外一组用于验证神经网络的实际效果。在随书代码中，读者应该可以很容易找到 zoo.90.percent.txt 和 zoo.10.percent.txt 这两个文件。其中，zoo.90.percent.txt 包含原始数据约 90%的数据量，用于神经网络的训练；zoo.10.percent.txt 则包含原始数据约 10%的数据量，用于神经网络实际效果的检验。这是一种很常见的验证手段，如果不这么做，或者说不区分训练数据集和测试数据集，则当测试的数据为训练数据的子集时，无法证明训练得到的网络模型具有普适性。即使拟合效果再好，也不能说明模型是好的，因为出现过拟合的可能性非常高。只有当测试数据集和训练集无关时，这种验证效果才能让人信服。因此，在本次案例中，使用约 90%的数据进行训练，另外约 10%的数据进行测试。

此外，还需要对原始数据进行预处理和编码，使之更适合神经网络的训练。在本例中，

主要表现在以下几个方面。

- 在实际训练数据中不再显示动物名称。很明显，动物名字对于分类没有任何帮助。
- 对动物的 legs 列，进行预处理，统一映射到 0～1 的浮点数。
- 分类信息 1～7 扩展为 7 列表示，而非 1 列。

对于第二点，原因很简单。对于动物特征的每一列，其特征的权重应该都是对等的。如果大部分数据的取值范围在 0～1，而某一列的取值范围跨度特别大，达到几百甚至几千，那么在网络训练时，这一列往往更容易占据主导地位，而这并不是我们希望看到的。因为某一列的权重，本质上来说与它的数值大小是无关的。比如，在员工信息中，员工年龄和员工年薪是非常重要的两个指标，但年龄通常在 20～70 岁，而年薪可能会是十几万甚至上百万元。那么当我们对这一组属性进行加权求和计算神经元的净输入时，很容易发现，年龄的大幅度变化会很容易地被年薪的小幅度变化所吞噬，从而影响训练效果。为了避免这个问题，通常的解决方案是，将所有特征的值域变化统一限制在 0～1。体现在这里，就是将动物的 legs 列的值映射到 0～1。这个操作叫作数据归一化。数据归一化的常用方法是 Max-Min 方法，其公式为：

$$\hat{X} = \frac{X - min}{max - min}$$

其中，X 为给定的属性，$\hat{X}$ 表示归一化后的结果，min 表示 X 属性的最小值，max 表示 X 属性的最大值。同上述公式，很容将任意的 X 值，映射到 0～1，从而使得训练数据在各个维度上处于相对一致的范围，改善训练的效果。

对于第三点，也是很常见神经网络处理技巧，因为神经元几乎很难准确输出 0 或者 1 这样的整数值，为了表达一个枚举数值，通常采用这种使用多个神经元的枚举方案。即，当预设种类数量为 7 时，就使用 7 个神经元作为输出，并且这 7 个神经元互斥，即只有一个神经元（通常是最活跃的那个）会被激活，其他的均被抑制。

因此，经过处理后，真正用于训练的数据格式如下：

```
0 0 1 0 0 1 1 1 1 0 0 1 0   1 0 0 0 0 0 1 0 0 0
1 0 0 1 0 0 0 1 1 1 0 0 0.5 0 1 0 0 0 0 0 0 0 1
0 0 1 0 0 0 1 0 0 0 0 0 0   0 0 0 1 0 0 0 0 0 0
0 0 1 0 0 1 1 0 0 0 0 0 0.5 0 0 0 1 0 0 0 0 0 0
```

其中，前 16 个字段表示训练用的动物特征，后 7 个字段（下画线部分）为分类结果。

7.3.3　代码实现

基于上述原理，即可构造一个用于 16 个输入神经元、7 个输出神经元的 BP 网络。这里定义中间隐层神经元数量为 6 个。

但在训练之前，我们首先需要一个能够读入训练文件格式并将其转为 DataSet 的工具方法：

```
01 public static List<DataSetRow> getTrainData(String filepath) throws IOException
{
02   InputStream in = AnimalClassify.class.getResourceAsStream(filepath);
03   BufferedReader br = new BufferedReader(new InputStreamReader(new DataInput
     Stream(in)));
04   List<DataSetRow> re = new Vector<DataSetRow>();
05   String line = null;
06   int Attribute_Len = 16;
07   while ((line = br.readLine()) != null) {
08         String[] item = line.split("\t");
09         double[] inputs = new double[Attribute_Len];
10         int i = 0;
11         for (i = 0; i < Attribute_Len; i++) {
12               inputs[i] = Double.parseDouble(item[i]);
13         }
14         double[] outputs = new double[7];
15         for (; i < Attribute_Len + 7; i++) {
16               outputs[i - Attribute_Len] = Double.parseDouble(item[i]);
17         }
18         re.add(new DataSetRow(inputs, outputs));
19   }
20    return re;
21 }
```

上述 getTrainData()方法读入训练文件，并通过文本解析，将每一行数据转为 DataSetRow，并最终返回 DateSetRow 集合。

接着就可以定义和训练神经网络了：

```
01 DataSet trainingSet = new DataSet(16, 7);
02 List<DataSetRow> rows = getTrainData("/geym/nn/mlperceptron/TS9010/zoo.90.percent.txt");
03 for (int i = 0; i < rows.size(); i++) {
```

```
04     trainingSet.addRow(rows.get(i));
05 }
06
07 MlPerceptron myMlPerceptron = new MlPerceptron(TransferFunctionType.SIGMOID, 16, 6, 7);
08 myMlPerceptron.getLearningRule().setMaxError(0.01d);
09 myMlPerceptron.getLearningRule().addListener(this);
10
11 System.out.println("Training neural network...");
12 myMlPerceptron.learn(trainingSet);
```

上述第 1 行定义了训练数据集，拥有 16 个输入和 7 个输出。第 2～5 行解析训练文件，并生成 DataSet。第 7 行定义了 BP 网络，拥有 16 个输入、6 个隐层神经元以及 7 个输出神经元。第 12 行正式训练网络。

接着，就可以使用测试集来检验网络的效果了：

```
01 public static void testNeuralNetwork(NeuralNetwork neuralNet) throws IOException
{
02   List<DataSetRow> rows = getTrainData("/geym/nn/mlperceptron/TS9010/zoo.10.
     percent.txt");
03   int count = 0, correctNumber = 0;
04   for (DataSetRow row : rows) {
05         neuralNet.setInput(row.getInput());
06         neuralNet.calculate();
07         double[] networkOutput = neuralNet.getOutput();
08         networkOutput = Utils.competition(networkOutput);
09         count++;
10         if (Arrays.equals(networkOutput, row.getDesiredOutput())) {
11               correctNumber++;
12         }
13    }
14    System.out.println("判断正确率:" + (correctNumber * 1.0 / count) * 100 + "%");
15 }
```

上述代码第 2 行，读入测试数据集 zoo.10.percent.txt。第 5～6 行对给定数据进行识别。第 7 行取得网络的实际输出。第 8 行根据竞争原理，将最活跃的那个神经元视为激活，其他神经元均为抑制状态，并根据这个激活神经元确定网络的最终判断。第 10 行将神经元的期望输出和实际判断结果进行比较，统计正确率。第 15 行打印 BP 网络在这个测试集上的正确率。

根据实验结果，这里的正确率可以达到 100%。

7.4 简单的语音识别

7.4.1 问题描述

BP 神经网络应用领域非常广泛，几乎所有能够被抽象为分类问题的场景都可以使用 BP 神经网络来处理。在本节中，将给出一个简单的语音识别场景。本节所需的数据可以从 https://archive. ics.uci.edu/ml/datasets/ISOLET 下载。当然，本书的随书代码中也已经附带了这组数据。

这组数据采集了 150 名受试者的口音。每名受试者将 26 个英文字母念 2 遍，即每名受试者产生 52 条训练数据。150 名受试者被分为 5 组，每组 30 人，数据以组的形式存放，分别是 isolet1，isolet2，isolet3，isolet4，isolet5。其中，第 1～4 组在这里训练中均作为训练数据，存放在文件 isolet1+2+3+4.data 中；isolet5 独立一个文件作为测试数据集。在整个数据集中，有 3 个样本因为某种原因丢失。

样本的数据格式非常简单，包含有 617 个特征列和 1 个标签列。特征列数据类型均是浮点数，用逗号分隔，标签列为整数，取值范围为 1～26。

7.4.2 代码实现

在训练神经网络之前，首先需要一段可以读取并解析输入文件的工具函数：

```
01 public static DataSet trainingData(String trainFile) throws FileNotFoundException,
   IOException {
02   unzipFileIfNotExist(trainFile);
03   DataSet ds = new DataSet(617, 26);
04   File fTrainFile = new File(trainFile);
05   BufferedReader br = new BufferedReader(new FileReader(fTrainFile));
06   String line = null;
07   while ((line = br.readLine()) != null) {
08         String[] strNumber=line.substring(0, line.length()-1).split(",");
09         double[] input = new double[strNumber.length-1];
10         double[] output = new double[26];
11         for(int i=0;i<strNumber.length-1;i++){
12               input[i]=Double.parseDouble(strNumber[i].trim());
13         }
14         output[Integer.parseInt(strNumber[strNumber.length-1].trim())-1]=1;
15         DataSetRow dr=new DataSetRow(input,output);
```

```
16          ds.addRow(dr);
17      }
18      return ds;
19  }
```

上述代码第 3 行定义了数据集格式，有 617 个输入和 26 个输出。与动物分类问题一样，这里使用 26 个神经元表示 26 个字母，所有神经元互斥，每次只有一个神经元被激活。第 8～16 行解析每个样本的训练数据。

接着进行网络训练：

```
01 DataSet trainingSet = IsoletReader.trainingData(IsoletDir +
"\\isolet1+2+3+4.data");
02 int inputCount=trainingSet.getRowAt(0).getInput().length;
03
04 MlPerceptron myMlPerceptron = new MlPerceptron(TransferFunctionType.SIGMOID,
   inputCount, 100,26);
05 // 设置可接受的误差
06 BackPropagation  learningRule = myMlPerceptron.getLearningRule();
07 learningRule.setLearningRate(0.05);
08 learningRule.setMaxError(0.04d);
09 learningRule.setMaxIterations(100);
10 learningRule.addListener(this);
11
12 System.out.println("Training neural network...");
13 myMlPerceptron.learn(trainingSet);
```

第 4 行定义了 BP 网络的结构，包括 617 个输入节点、100 个隐层节点和 26 个输出节点。第 7～9 行设置了训练的一些基本参数，在这里学习速率为 0.05，最大可接受误差为 0.04，最大迭代次数为 100 次。第 13 行开始执行训练。

训练完成后，就可以使用第 5 组数据 isolet5.data 进行测试：

```
01 public static void testNeuralNetwork(NeuralNetwork neuralNet) throws IOException
{
02   DataSet testDataSet = IsoletReader.trainingData(IsoletDir + "\\isolet5.data");
03
04   int rightCount = 0;
05   int i=0;
06   for (; i < testDataSet.size(); i++) {
07         neuralNet.setInput(testDataSet.getRowAt(i).getInput());
08         neuralNet.calculate();
09         double[] networkOutput = neuralNet.getOutput();
```

```
10          // 将活跃度最高的输出视为 1，其余视为 0
11          networkOutput = Utils.competition(networkOutput);
12          if (Arrays.equals(networkOutput,
testDataSet.getRowAt(i).getDesiredOutput())) {
13              rightCount++;
14          }
15          if(i%1000==0)
16              System.out.println("正确率:" + rightCount * 1.0 / (i+1));
17      }
18      System.out.println("正确率:" + rightCount * 1.0 / (i+1));
19  }
```

上述代码第 2 行读入测试数据集。第 6～17 行对每一个样本进行预测，并与期望数据进行比较，同时统计其正确率。最终，BP 神经网络在这个案例中的最终正确率保持在 94% 左右。

有兴趣的读者还可以参考随书代码中的 IsoletTraining 和 IsoletReader 类，它们给出了完整的可执行的程序。

7.5 MNIST 手写体识别

7.5.1 问题描述

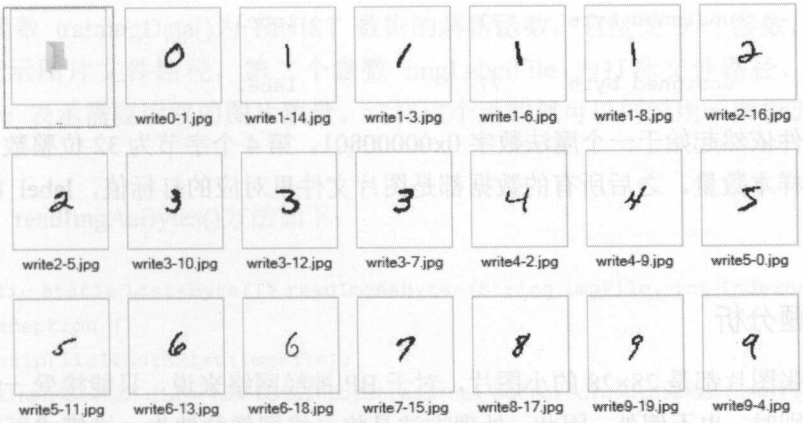
手写体识别一直是人工智能中一个非常重要的领域。前面部分已经介绍了使用 ADALINE 网络进行印刷体数字识别的方法。与印刷体相比，手写体识别的难度会更高，在本节中，我们将使用非常经典的 MNIST 手写体字符集，对 BP 网络进行训练与测试。

MNIST 字符集可以从 http://yann.lecun.com/exdb/mnist/处下载。在本书的随机代码中，也已经附带了该数据集。该数据集包含有 60000 条手写体训练数据以及 10000 条测试数据。

MNIST 字符集主要由 4 个文件组成。

- train-images-idx3-ubyte.gz：用于训练的 60000 张图像（9912422 bytes）。
- train-labels-idx1-ubyte.gz：用于训练的图像打标数据（28881 bytes）。
- t10k-images-idx3-ubyte.gz：用于测试的 10000 张图像（1648877 bytes）。
- t10k-labels-idx1-ubyte.gz：用于测试的图像打标数据（4542 bytes）。

图 7-7 所示是一些从 MNIST 数据集中提取的部分手写体图片。



▲图 7-7 部分 MNIST 手写体图片

可以看到，手写体图片的多样性和变化度都不是印刷体可以比拟的。

MNIST 数据集有自己固定的文件格式，并不是简单的图片压缩打包，其图片部分（train-images-idx3-ubyte 和 t10k-images-idx3-ubyte）的格式如下：

```
[offset] [type]          [value]          [description]
0000     32 bit integer  0x00000803(2051) magic number
0004     32 bit integer  60000            number of images
0008     32 bit integer  28               number of rows
0012     32 bit integer  28               number of columns
0016     unsigned byte   ??               pixel
0017     unsigned byte   ??               pixel
........
xxxx     unsigned byte   ??               pixel
```

其中，offset 表示文件的偏移量，type 表示这个偏移量上的数据类型，它可以是 32 位整数或者 8 位无符号整数。文件前 4 个字节为魔法数字，固定为 0x00000803。第 4 个字节上为 32 位整数，表示该文件中的数据总量。第 8 个字节和第 12 个字节均是 32 位整数，表示每张图片包含的行和列的长度（以像素为单位）。第 16 个字节开始为每个像素的具体数值。

打标数据的格式如下（train-labels-idx1-ubyte 和 t10k-labels-idx1-ubyte）：

```
[offset] [type]          [value]          [description]
0000     32 bit integer  0x00000801(2049) magic number (MSB first)
0004     32 bit integer  60000            number of items
0008     unsigned byte   ??               label
```

```
0009          unsigned byte     ??               label
........
xxxx          unsigned byte     ??               label
```

打标文件依然起始于一个魔法数字 0x00000801。第 4 个字节为 32 位整数，表示该文件中描述的样本数量。之后所有的数据都是图片文件里对应的打标值，label 的取值范围为 0～9。

7.5.2　问题分析

由于每张图片都是 28×28 的小图片，对于 BP 神经网络来说，只能接受一维输入。在进行图像识别时，也不例外，因此，处理方法是将二维图像拉伸为一维模式再传递给神经网络，即将 28×28 的二维数组看成是长度为 784 的一维数组，所以此处的 BP 神经网络需要有 784 个神经元输入，每个神经元对应图片中的一个像素。

此外，每个像素值的范围为 0～255。过于丰富的输入并不利于神经网络的学习，因此在提取图片时，统一将像素值小于 128 的像素点设为 0，将像素值大于 128 的像素点设为 1，从而简化网络的输入。

对于网络的输出，由于只可能是 0～9 的数字，因此可以使用 10 个神经元作为输出。

综上所述，就可以在此构造一个拥有 784 个输入神经元、100 个中间隐层神经元以及 10 个输出神经元的 BP 神经网络。

7.5.3　代码实现

在训练神经网络之前，首先需要读入相关数据，并将其整理为 DataSet 格式：

```
1 public static DataSet trainingData(String imgFile, String imgLabelFile, int indexBefore)
  throws IOException {
2       List<byte[]> inputs = readImgAsBytes(imgFile, indexBefore);
3       byte[] labels = readImgLabel(imgLabelFile, indexBefore);
4       DataSet ds = new DataSet(inputs.get(0).length, 10);
5       for (int i = 0; i < inputs.size(); i++) {
6             ds.addRow(encodeInput(inputs.get(i)), encodeOutput(labels[i]));
7       }
8       return ds;
9 }
```

上述函数 trainingData()为 MNIST 数据的解析函数，它接受 3 个参数，第一个参数 imgFile 表示图片文件路径，第二个参数 imgLabelFile 为打标文件路径，第三个参数 indexBefore 表示需要读取的图片数量。这样这个函数就可以同时用于解析训练数据和测试数据。

其中，readImgAsBytes()方法如下：

```
01 public static List<byte[]> readImgAsBytes(String imgFile, int indexBefore) throws
   IOException {
02   unzipFileIfNotExist(imgFile);
03 DataInputStream dis = new DataInputStream(new BufferedInputStream(new FileInputStream
   (new File(imgFile))));
04      try {
05         ....省略部分非核心代码
06            int rows = dis.readInt();
07            int cols = dis.readInt();
08            List<byte[]> re = new ArrayList<byte[]>(indexBefore);
09            for (int i = 0; i < indexBefore; i++) {
10                  byte[] bImg = new byte[rows * cols];
11                  dis.read(bImg);
12                  for (int k = 0; k < bImg.length; k++) {
13                        if ((bImg[k] & 0xff) < 128) {
14                              bImg[k] = 0;
15                        } else {
16                              bImg[k] = 1;
17                        }
18                  }
19                  re.add(bImg);
20            }
21            return re;
22      } finally {
23            dis.close();
24      }
25 }
```

上述代码读入一个 MNIST 文件，并将前 indexBefore 个数据进行返回。第 13～17 行将像素值进行二值化判断，即像素值小于 128 的统一设置为 0，大于等于 128 的统一设置为 1。每张图片对应一个 byte 数组。

类似的方法可以实现读取打标数据，下面是 readImgLable()的核心代码：

```
01 public static byte[] readImgLabel(String imgLabelFile, int indexBefore) throws
   IOException {
02   unzipFileIfNotExist(imgLabelFile);
03 DataInputStream dis=new DataInputStream(new BufferedInputStream(new FileInputStream
   (new File(imgLabelFile))));
04       try {
05          ....省略部分非核心代码
06             byte[] lables = new byte[indexBefore];
07             dis.read(lables);
08             return lables;
09       } finally {
10             dis.close();
11       }
12 }
```

由于打标文件中每一个样本只使用了 1 个字节，因此这里简单地读入前 indexBefore 个字节数即可。

有了训练用的 DataSet 后，就可以开始训练神经网络了：

```
01 DataSet trainingSet = MnistReader.trainingData(MnistDir + "\\train-images.idx3-ubyte",
02       MnistDir + "\\train-labels.idx1-ubyte", 60000);
03
04 MlPerceptron myMlPerceptron = new MlPerceptron(TransferFunctionType.SIGMOID, 784,
   100,10);
05 // 设置可接受的误差
06 BackPropagation  learningRule = myMlPerceptron.getLearningRule();
07 learningRule.setLearningRate(0.05);
08 learningRule.setMaxError(0.001d);
09 learningRule.setMaxIterations(3);
10 learningRule.addListener(this);
11
12 System.out.println("Training neural network...");
13 myMlPerceptron.learn(trainingSet);
```

上述代码第 4 行，定义了一个有 784 个输入神经元，100 个隐层神经元和 10 个输出神经元的 BP 神经网络。第 7～9 行分别设置了网络的学习速率为 0.05，最大误差为 0.001，最大迭代次数为 3 次。由于在本例中，已经含有 60000 条训练样本，再加上网络的神经元数量比较多，导致每一次迭代都需要占用不少时间，所以，在这里将迭代次数设置得比较小。第 13 行开始执行学习算法。

当网络训练完成后，就可以进行学习效果的测试了：

```
01 public static void testNeuralNetwork(NeuralNetwork neuralNet) throws IOException
{
02   DataSet testDataSet = MnistReader.trainingData(MnistDir + "\\t10k-images.idx3-ubyte",
03            MnistDir + "\\t10k-labels.idx1-ubyte", 10000);
04   int rightCount = 0;
05   int i=0;
06   for (; i < testDataSet.size(); i++) {
07         neuralNet.setInput(testDataSet.getRowAt(i).getInput());
08         neuralNet.calculate();
09         double[] networkOutput = neuralNet.getOutput();
10         // 将活跃度最高的输出设为 1，其余设为 0
11         networkOutput = Utils.competition(networkOutput);
12         if (Arrays.equals(networkOutput, testDataSet.getRowAt(i).getDesiredOutput())) {
13               rightCount++;
14         }
15         if(i%1000==0)
16              System.out.println("正确率:" + rightCount * 1.0 / (i+1));
17   }
18   System.out.println("正确率:" + rightCount * 1.0 / (i+1));
19 }
```

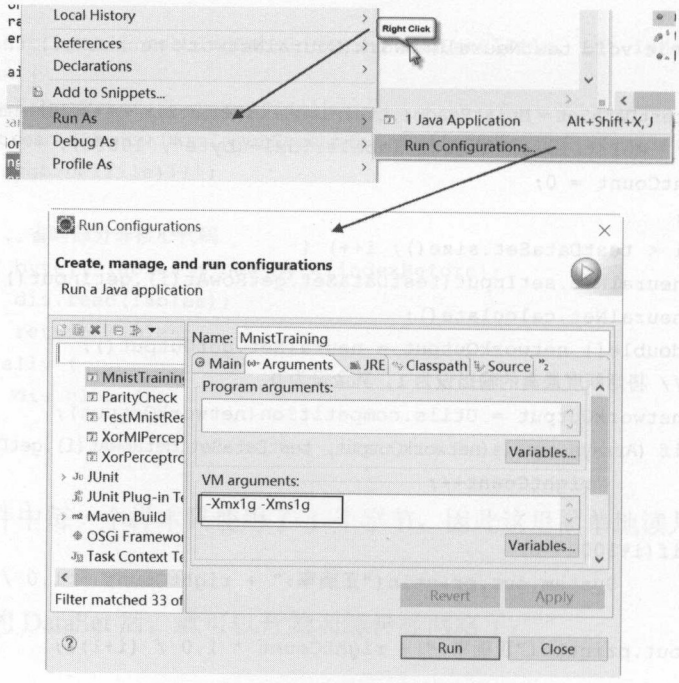
上述代码第 2～3 行读入 10000 条测试数据。第 7～9 行进行手写体识别，第 12 行将网络的识别结果和期望结果进行比对，同时在第 13 行统计正确的数量。最后，在第 18 行输出对这 10000 条数据识别的正确率。

由于在本案例中，数据规模比较庞大，因此对内存有一定要求。如果读者直接运行这段训练代码，很可能会遇到如下问题：

```
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
    at geym.nn.mnist.MnistReader.encodeInput(MnistReader.java:161)
    at geym.nn.mnist.MnistReader.trainingData(MnistReader.java:155)
    at geym.nn.mnist.MnistTraining.run(MnistTraining.java:24)
    at geym.nn.mnist.MnistTraining.main(MnistTraining.java:20)
```

这是因为在训练过程中所需的堆内存空间已经超过默认情况下 Java 使用的堆空间大小。如果读者在训练神经网络的过程中遇到同样问题，可以通过指定堆空间大小来修复这个问题。图 7-8 所示显示了在 Eclipse 中将 Java 的堆空间大小设置为 1GB。

经过多次测试，最终的正确率保持在 94%左右，达到了令人满意的结果。有兴趣的读者可以参考随书代码中的 MnistTraining 类自行测试。



▲图 7-8　设置 Java 的运行时堆空间大小

7.6 总结

本章主要介绍了一些 BP 神经网络的使用案例。作为一款经典的监督学习算法，BP 网络可以很好地应用在预测、分类等领域。本章所阐述的 5 个应用场景包括数字的奇偶性判断、函数逼近、动物分类实验、语音识别和 MNIST 手写体识别。虽然应用场景各不相同，但均能够使用 BP 神经网络并使用类似的同一套方案解决，由此说明 BP 网络应用具有多样性和普适性，不愧于是现代最为流行的神经网络之一。

第8章 Hopfield神经网络

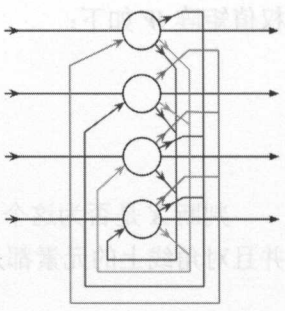
Hopfield神经网络是一种与多层感知机完全不同的神经网络。对于多层感知机而言，网络的层次非常清晰，数据总是从前一层流向下一层，同一层之间的神经元并无连接。而Hopfield网络并没有分层，或者说它只有简单的一层，并且在所有神经元之间建立全连接。

Hopfield神经网络是美国加州工学院物理学家J. J. Hopfield于1982年提出的，可以用于联想记忆、最优化问题求解等领域。Hopfield网络又可以分为离散型网络和连续性网络。本章将主要介绍离散型网络在联想学习中的应用。

8.1 Hopfield神经网络的结构和原理

8.1.1 Hopfield网络的结构

离散型Hopfield网络的结构如图8-1所示。



▲图8-1 包含4个神经元的Hopfield网络

可以看到，严格意义上说，网络只有一层，但是在这一层中，所有的神经元均两两连接。网络从一个输入向量开始计算后，每次迭代时，总是把上一次的输出作为下一次的输入。离散型Hopfield网络的神经元传输函数通常为符号函数*sgn*。单个神经元不接受自反馈，即$w_{ii}=0$。同时，网络的连接权重通常是对称的，即$w_{ij}=w_{ji}$，也就是神经元i到神经元j的连接权值和神经元j到神经元i的连接权值是相等的。权值矩阵的对称性，是网络能够收敛到吸引子的一个非常重要的前提条件。

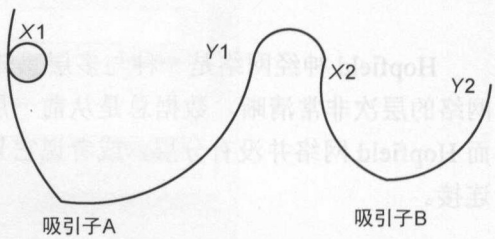
8.1.2　网络吸引子

对于离散 Hopfield 网络来说，在进行这种反馈迭代后，可能出现两种可能：

（1）网络维持在一种稳定的状态，即每次的输出总是等于输入。

（2）网络出现自持振荡，即虽然网络无法达到稳定状态，但陷入一个状态列表循环。

Hopfield 网络可以实现存储若干个稳定点，即网络的稳定状态可以不止一个。而 Hopfield 网络也正是依靠这些稳定点来进行联想学习的。当网络达到稳定状态 $\boldsymbol{S}$ 时，这个状态也称为网络吸引子。吸引子这个称呼正是一种很形象的比喻，因为网络一旦落入这个状态，就再也无法逃脱了。图 8-2 所示显示了网络吸引子的一个形象比喻。当小球下落后，会在低谷处来回振荡，但最终会在最低处静止不动，那个最低的位置就如同网络吸引子。



▲图 8-2　从物理学上看网络吸引子

能够使得网络稳定在同一吸引子的所有初始状态的集合称为吸引域。一个有意义的吸引子必须具有一定的吸引域，只有这样，网络才可能收敛到那个吸引域。如图 8-2 所示，吸引子 A 的吸引域为 $X1 \sim Y1$，吸引子 B 的吸引域为 $X2 \sim Y2$。为了使网络的联想功能尽可能正确，目标状态必须是一个吸引子，并且其吸引域也要足够大。否则，就极有可能出现对样本稍加干扰，就使得网络无法进行正确联想的情况，从而降低网络的识别正确率。

下面用一个简单的例子，说明网络吸引子的特性。

假设有一个离散 Hopfield 网络，节点数量 n=3，为简单起见，令偏置为 0，状态 $\boldsymbol{X}$、权值矩阵 $\boldsymbol{W}$ 如下：

$$\boldsymbol{X}=\begin{pmatrix}1\\1\\1\end{pmatrix}\quad \boldsymbol{W}=\begin{pmatrix}0&2&2\\2&0&2\\2&2&0\end{pmatrix}$$

判断 $\boldsymbol{X}$ 是否为这个网络的吸引子？注意，通常 Hopfield 网络的权值矩阵就是对称的，并且对角线上的元素都是 0。

$$f(\boldsymbol{WX})=f\begin{pmatrix}4\\4\\4\end{pmatrix}=\begin{pmatrix}sgn(4)\\sgn(4)\\sgn(4)\end{pmatrix}=\begin{pmatrix}1\\1\\1\end{pmatrix}$$

很显然，对于输入 $\boldsymbol{X}$，网络的输出依然是 $\boldsymbol{X}$，因此 $\boldsymbol{X}$ 为这个 Hopfield 网络的吸引子。

现在，假设有一初始输入 $\boldsymbol{Y}$ 为 $\begin{pmatrix}-1\\1\\1\end{pmatrix}$。则在一次迭代过程中，首先调整第一个神经元的输出为 $sgn(1\times2+1\times2)=1$，继而调整第二个神经元输出为 $sgn(1\times2+1\times2)=1$，同理第三个神经元输出依然是 $sgn(1\times2+1\times2)=1$，因此，状态 $\boldsymbol{Y}$ 经过一次状态转换后，落入吸引子 $\begin{pmatrix}1\\1\\1\end{pmatrix}$。

8.1.3 网络权值的设计

吸引子是 Hopfield 网络正常工作的关键部分。而决定吸引子的，却是 Hopfield 网络的权值以及神经元的阈值。因此，Hopfield 网络的权值设计对于网络的运行能力有着非常重要的作用。对于小规模的 Hopfield 网络来说，可以使用联立方程的方法来求解网络权值。

为简单起见，我们在这里假设神经元的阈值为 0。在这个前提下，我们需要设计一个 Hopfield 网络，让它拥有吸引子 $\begin{pmatrix}1\\1\\1\end{pmatrix}$。

首先，对于任何 Hopfield 网络，必须有 $w_{ij}=w_{ji}$。对于吸引子 $\begin{pmatrix}1\\1\\1\end{pmatrix}$，各节点的净输入必须满足以下条件：

$$\begin{aligned}\text{net}_1&=w_{12}\times1+w_{13}\times1>0\\\text{net}_2&=w_{12}\times1+w_{23}\times1>0\\\text{net}_3&=w_{13}\times1+w_{23}\times1>0\end{aligned}$$

联立以上三个方程，可以得到权重必须满足的条件是：

$$\begin{aligned}w_{12}&>-w_{13}\\w_{12}&>-w_{23}\\w_{13}&>-w_{23}\end{aligned}$$

任取满足条件的 w 值，这里取：

$$w_{12}=2;\ w_{13}=1;\ w_{23}=1$$

因此，构建的网络权重为：

$$\boldsymbol{W}=\begin{pmatrix}0 & 2 & 1\\ 2 & 0 & 1\\ 1 & 1 & 0\end{pmatrix}$$

验证 $\boldsymbol{W}$ 是否为吸引子，只需要代入 $\begin{pmatrix}1\\1\\1\end{pmatrix}$：

$$Sgn\left(\begin{pmatrix}0 & 2 & 1\\ 2 & 0 & 1\\ 1 & 1 & 0\end{pmatrix}\begin{pmatrix}1\\1\\1\end{pmatrix}\right)=\begin{pmatrix}Sgn(3)\\Sgn(3)\\Sgn(2)\end{pmatrix}=\begin{pmatrix}1\\1\\1\end{pmatrix}$$

很显然，在吸引子比较少的情况下，使用这种方法还勉强可以求解。但是，如果网络规模比较大，需要设计的吸引子数量很多，这种方法自然就行不通了。

一种更为通用的权值设计方法，是采用 Hebb 规则的外积和法。这种方法的描述如下：

假设有 N 个学习样本 X^1，X^2，…，X^N，样本两两正交（通俗地讲就是垂直），并且样本的维度大于样本的数量，那么权值设计公式为：

$$w_{ij}=\begin{cases}\sum_{n=1}^{N}x_i^n x_j^n & i\neq j\\ 0 & i=j\end{cases}$$

简单来说，神经元 i 和神经元 j 的权值等于每一个训练样本上 i 分量和 j 分量的乘积之和。使用这个公式构造的权值，可以保证给定样本是网络的吸引子。但是网络依然还是有可能存在其他吸引子，很显然，这些额外的吸引子并非对我们有意义，通常会把网络带向错误的方向。

下面以一个简单的例子来说明网络权值的构造：

假设需要记忆 $\boldsymbol{X}=\begin{pmatrix}1\\1\\-1\\-1\end{pmatrix}$，$\boldsymbol{Y}=\begin{pmatrix}-1\\-1\\-1\\-1\end{pmatrix}$ 两组模式，显示 2 条样本正交，并且维度 4 大于样本数量 2。

根据外积和法可以求得：

$$w_{12}=1\times1+(-1)\times(-1)=2$$

$$w_{13}=1\times(-1)+(-1)\times(-1)=0$$
$$w_{14}=1\times(-1)+(-1)\times(-1)=0$$
$$w_{23}=1\times(-1)+(-1)\times(-1)=0$$
$$w_{24}=1\times(-1)+(-1)\times(-1)=0$$
$$w_{34}=-1\times(-1)+(-1)\times(-1)=2$$

因此，有：

$$\boldsymbol{W}=\begin{pmatrix}0&2&0&0\\2&0&0&0\\0&0&0&2\\0&0&2&0\end{pmatrix}$$

检验结果是否正确，只需要计算 $\boldsymbol{W}$ 与模式的乘积即可：

$$f(\boldsymbol{WX})=f\begin{pmatrix}2\\2\\-2\\-2\end{pmatrix}=\begin{pmatrix}sgn(2)\\sgn(2)\\sgn(-2)\\sgn(-2)\end{pmatrix}=\begin{pmatrix}1\\1\\-1\\-1\end{pmatrix}=\boldsymbol{X}$$

$$f(\boldsymbol{WY})=f\begin{pmatrix}-2\\-2\\-2\\-2\end{pmatrix}=\begin{pmatrix}sgn(-2)\\sgn(-2)\\sgn(-2)\\sgn(-2)\end{pmatrix}=\begin{pmatrix}-1\\-1\\-1\\-1\end{pmatrix}=\boldsymbol{Y}$$

由此可见，只需要根据外积和法构造的权重，就能将给定样本作为 Hopfield 网络的吸引子。

8.2 网络的存储容量

虽然 Hopfield 网络具有记忆模式的能力，但是在网络规模一定时，其能够记忆的模式数据是有限的。对于离散 Hopfield 网络来说，假设网络包含 n 个神经元，那么它可以记忆的最大模式为 n。能够被网络记忆的模式也正是网络的吸引子。对于给定若干条正交样本，只要按照外积和法求解得到 Hopfield 网络，这个 Hopfield 网络的吸引子就是这些正交样本。因此，为了让网络记忆一组模式，只需要按照外积和法求得网络的权值即可。但不幸的是，在实际情况下，模式样本不可能满足两两正交的条件，对于非正交的样本，网络的信息存储容量会大大降低。当样本非正交时，每训练一组模式都有可能会影响已经记忆的模式，随之训练样本的增加，权值不断变化，各个样本之间也会相互影响，最终网络可能

遗忘已经记住的模式。

因此，对于给定规模的 Hopfield 网络，要记忆的模式越多，联想出错的可能性就越大。实际情况下，对于规模为 n 的网络，一般只能存储 $0.15n$ 个样本。

8.3 Hopfield 神经网络的 Java 实现

前文中，已经比较详细地介绍了有关 Hopfield 网络的理论知识。现在，我们已经具有足够多的信息来实现一个 Hopfield 网络了。在 Neuroph 中实现 Hopfield 网络的步骤和多层感知机类似，都可以从 NeuralNetwork 类继承并加以扩展。

8.3.1 Hopfield 网络构造函数

构造 Hopfield 网络时，只需要一个最关键的参数，即网络包含神经元的数量，因此 Hopfield 网络的构造函数只需要传入一个网络规模的参数即可：

```
01 public class Hopfield extends NeuralNetwork {
02   public Hopfield(intneuronsCount) {
03
04        // init neuron settings for hopfield network
05        NeuronPropertiesneuronProperties = new NeuronProperties();
06        neuronProperties.setProperty("neuronType", InputOutputNeuron.class);
07        neuronProperties.setProperty("bias", new Double(0));
08        neuronProperties.setProperty("transferFunction", TransferFunctionType.SGN);
09
10        this.createNetwork(neuronsCount, neuronProperties);
11   }
```

上述代码定义了 Hopfield 网络的构造函数。其传入参数为 neuronsCount，表示网络的规模。在第 6 行，指定网络的每一个神经元均为 InputOutputNeuron；第 7 行设置了神经元偏置为 0；第 8 行指定神经元的传输函数为符号函数 *Sgn*。

构造函数中的 createNetwork()函数实现如下：

```
1 private void createNetwork(intneuronsCount, NeuronPropertiesneuronProperties) {
2     this.setNetworkType(NeuralNetworkType.HOPFIELD);
3     Layer layer = LayerFactory.createLayer(neuronsCount, neuronProperties);
4     ConnectionFactory.fullConnect(layer, 0.1);
5     this.addLayer(layer);
```

```
6     NeuralNetworkFactory.setDefaultIO(this);
7     this.setLearningRule(new StandHopfieldLearning());
8 }
```

可以看到，Hopfield 网络的整体构造步骤和多层感知机大体相似，都是通过 LayerFactory 先构造网络的层，然后设置网络中的神经元连接，最后设置学习算法。但值得注意的是，第 4 行的连接设置和多层感知机是完全不同的。在多层感知机中，创建的连接是两层之间的神经元两两互联，同一层中的神经元没有连接。但在此处，则是在同一层中的神经元两两互联。

8.3.2 Hopfield 网络的神经及其特点

Hopfield 网络的神经元比较特殊，使用的是 InputOutputNeuron。InputOutputNeuron 神经元是 Neuron 的子类，它对普通的 Neuron 神经元进行了扩展，主要进行了两个方面的增强。

第一，对于普通的 Neuron 神经元，并不会设置神经元偏置。根据第 4 章的介绍已经知道，为了确保计算的统一性，神经元偏置都被上层的贝叶斯神经元 BiasNeuron 取代，这样就可以使用同样的算法处理偏置和权值连接。但在 Hopfield 网络中，情况发生了变化。Hopfield 网络只有一层，因此不存在可以放置贝叶斯神经元的地方，使得神经元偏置无法使用这种方式简化处理。所以，InputOutputNeuron 的一大扩展作用就是支持对偏置的设置。

第二，由于网络只有一层，甚至没有输入层（实际上这一层既是输入层同时也是输出层），因此 Hopfield 网络的神经元应该同时具备输入层神经元的功能和输出层神经元的功能。即，在神经元初次接受数据输入时，应该表现得像一个输入神经元，网络的净输入应该就是外部输入，而在后续的反馈迭代训练中，网络的净输入应该由输入函数和神经元的连接权值得到。

以下代码是对上述两点的实现：

```
01 public void calculate() {
02
03   if (!externalInputSet) {
04         if (this.hasInputConnections())
05               netInput = inputFunction.getOutput(this.inputConnections);
06   }
07
```

```
08    // calculqate cell output
09    this.output = transferFunction.getOutput(this.netInput + bias);
10
11    if (externalInputSet) {
12          externalInputSet = false;
13          netInput = 0;
14    }
15 }
```

其中，第 3 行的 externalInputSet 变量表示当前神经元的工作状态，即当前神经元是直接接受外部输入，还是处理后续的反馈迭代训练中。如果作为简单的输入神经元，则不会进行第 5 行的计算，而是直接将外部输入进行传输函数计算，并输出。反之，则进行第 5 行的反馈信息，计算得到自身的净输入。第 9 行的 bias 即为当前神经元的偏置，可以在神经元上直接通过 setBias()函数设置。

8.3.3　Hopfield 网络学习算法

在本例给出的 Hopfield 网络中，指定的学习算法为 StandHopfieldLearning。它正是前面已经介绍的外积和法的完整实现：

```
01 public void learn(DataSettrainingSet) {
02    int M = trainingSet.size();
03    int N = neuralNetwork.getLayerAt(0).getNeuronsCount();
04    Layer hopfieldLayer = neuralNetwork.getLayerAt(0);
05
06    for (int i = 0; i < N; i++) {
07          for (int j = 0; j < N; j++) {
08                if (j == i)
09                      continue;
10                Neuron ni = hopfieldLayer.getNeuronAt(i);
11                Neuron nj = hopfieldLayer.getNeuronAt(j);
12                Connection cij = nj.getConnectionFrom(ni);
13                Connection cji = ni.getConnectionFrom(nj);
14
15                double wij=0;
16                for(int k = 0;k <M;k++){
17                      DataSetRow row=trainingSet.getRowAt(k);
18                      double[] inputs=row.getInput();
19                      wij+=inputs[i]*inputs[j];
20                }
21                cij.getWeight().setValue(wij);
```

```
22                cji.getWeight().setValue(wij);
23            }// j
24    } // i
25 }
```

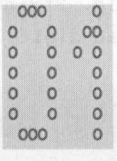
根据外积和法，神经元的权值应该是各个样本上对应的分量的乘积的累加。上述代码中，样本数量为 M，神经元数量为 N。第 8 行，设置权值矩阵对角线元素为 0。第 16～20 行，将 M 条样本对应分量上的数值相乘后进行求和，即外积和法的核心公式实现。第 21～22 行，根据权值对称性设置 w_{ij} 和 w_{ji} 的值。

8.4 Hopfield 网络还原带有噪点的字符

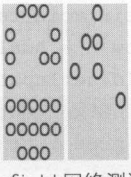
第 5 章曾经使用 ADALINE 网络识别印刷体数字，实验结果是：ADALINE 不仅可以识别无干扰的印刷体，而且可以识别带有噪点的印刷体字符。在这里，让我们进行一个类似的实验，即让 Hopfield 网络还原带有噪点的字符。但与 ADALINE 网络不同的是，ADALINE 网络的学习属于监督式学习，即我们将给出每个图像的真实含义，ADALINE 网络将会把“看到”的图像还原为它们的真实含义。但 Hopfield 网络不同，Hopfield 网络的学习算法属于非监督式学习。Hopfield 网络并没有学习指导数据，它并不真的知道这些数字背后的真实含义，而只是单纯地“记住”了这些图像。当图像受到干扰时，它只是回忆起原先记住的与之最接近的那个图像而已。

为简化起见，在本例中，我们只使用 0 和 1 两个数字进行试验，如图 8-3 所示。

我们将上述两个数字作为 Hopfield 网络的训练数据。注意，这里不存在打标数据，Hopfield 网络在这里记住的是图像本身，当 Hopfield 网络记住这两个字符后，给出图 8-4 所示的两个带有噪点的图像。



▲图 8-3 Hopfield 网络训练数字



▲图 8-4 Hopfield 网络测试用数字图像

很明显，可以看到测试用的 0 存在大量的噪点，几乎达到了整个字符的一半。而测试用的 1 则是有一半的缺失和部分噪点。在这种大量变形和损坏的情况下，Hopfield 网络还可以将它们还原成干净的数字图像吗？让我们一起实验吧！有兴趣的读者可以在随书代码

中找到 HopfieldSample 类参考实现。

首先，我们需要构造训练数据：

```
DataSettrainingSet = new DataSet(35);
trainingSet.addRow(AdalineDemo.createTrainDataRow(DIGITS[0], 0)); // 0
trainingSet.addRow(AdalineDemo.createTrainDataRow(DIGITS[1], 0)); // 1
```

变量 DIGITS 中存放的就是原始图像 0 和 1。字符图像是 5×7 的规格，共 35 个像素，所以 DataSet 的大小也是 35。由于图像的规格大小和格式与 ADALINE 网络实验时完全一致，因此在将图像转为可训练数组的过程中，沿用了 ADALINE 的转换函数。其次，由于 Hopfield 网络不需要打标的训练数据，因此 AdalineDemo.createTrainDataRow()函数的第二个参数，即期望值在这里没有实际意义，可以忽略不计，所以统一传入 0。

接着，定义一个与输入数据大小一致的 Hopfield 网络，将这些训练数据传入该网络：

```
Hopfield myHopfield = new Hopfield(35);
myHopfield.learn(trainingSet);
```

上述代码第 2 行的 learn()方法，将执行 Hopfield 网络的学习过程，也就是使用外积和法重新定义权值。

在固定网络权值后，就可以将测试用的污损字符传递给网络：

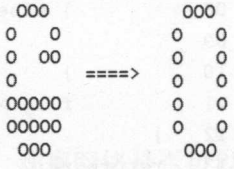
```
for (int i = 0; i <= 1; i++) {
      recallDigit(myHopfield, BAD_DIGITS[i]);
}
```

上述代码中 recallDigit()即回忆函数，它将把污损字符进行还原。它的第一个参数是已经训练好的 Hopfield 网络本身；第二个参数是需要还原的字符。它的实现如下：

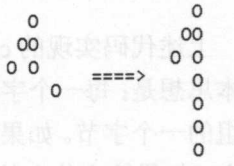
```
01 private static void recallDigit( Hopfield myHopfield, String[] bad_digit) {
02    DataSetRow h = AdalineDemo.createTrainDataRow(bad_digit, 0);
03    myHopfield.setInput(h.getInput());
04    double[] networkOutput = null;
05    double[] preNetworkOutput = null;
06    while (true) {
07          myHopfield.calculate();
08          networkOutput = myHopfield.getOutput();
09          if (preNetworkOutput == null) {
10                preNetworkOutput = networkOutput;
11                continue;
```

```
12          }
13          if (Arrays.equals(networkOutput, preNetworkOutput)) {
14              break;
15          }
16          preNetworkOutput = networkOutput;
17      }
18
19      System.out.println("Input: ");
20      printDigit(h.getInput());
21      System.out.println(" Output ====>" );
22      printDigit(networkOutput);
23  }
```

上述代码第 2 行，读入测试数据，其中 bad_digit 即为污损字符。第 3 行设置网络的起始状态。第 6～17 行为网络的振荡迭代，它将一直持续进行，直到网络达到稳定状态。网络的稳定状态由第 13 行表示，即网络的第 n 次输出，与第 n–1 次输出相同，表示网络达到稳定。网络达到稳定的另一层含义就是网络已经落入某一个吸引子，根据外积和法的规则，这个吸引子可能就是标准的字符图像 0 或者 1，当然也可能是其他无意义的吸引子。因此，在这里我们期望的场景就是，网络落入的吸引子就是训练图像本身，否则代表着回忆失败。



网络稳定后，在第 22 行输出网络的最终状态。第 20 行输出的是网络的初态。执行程序后，输出如图 8-5（篇幅问题对输出格式略微进行调整）。



▲图 8-5　受损字符的还原过程

可以看到，这个 Hopfield 网络已经将受损的字符进行了还原，完全达到我们的预期。

本节最后，留给读者一个思考，如果希望 Hopfield 网络记忆全部 10 个数字的话，当前网络可用吗？（可以参考本书 8.2 节“网络的存储容量”中内容）

8.5 Hopfield 网络的自联想案例

基于 Hopfield 的自联想功能，还能实现一些简单有趣的小功能。在本例中，将给大家展示如何使用 Hopfield 网络进行英文单词的自动纠正。打个比方，如果把单词 bad 误写成 bbd，那么 Hopfield 网络应该可以将 bbd 这个错误的单词重新还原成 bad。在本节中，就

将实现这样一个程序。

对于单词联想来说，我们只需要把目标单词设计为网络的吸引子，那么很自然，如果 Hopfield 网络得到一个任意的单词输入，就很可能将这个单词回忆成作为吸引子的一个单词，如果恰好这个单词就是我们的期望输出，那么“回忆”就成功了。

因此，这里的关键一点就在于，如何将一个单词编码成网络可用的输入，也就是一个数组。这里采用的方案是将单词拆分为字符，将每一个字符的 ASCII 码进行提取后编码成网络可用数组。下面是一段将字符编码为数组的方案：

```
01 public static double[] char2Doubles(char c) {
02   intintChar = (int) c;
03   double[] re = new double[8];
04   int flag = 0x80;
05   for (int i = 0; i <re.length; i++) {
06         if ((flag &intChar) != 0) {
07               re[i] = 1;
08         } else {
09               re[i] = -1;
10         }
11         flag >>= 1;
12   }
13   return re;
14}
```

上述代码实现的 char2Doubles()函数，将给定的字符转为一个长度为 8 的 double 数组。基本思想是：每一个字符的 ASCII 码对应的 byte 整数由 8 位组成，将每一位对应到 double 数组的一个字节。如果给定位上的值是 1，那么对应的 double 数组中的那个字节就设为 1；反之，如果给定位上的值是 0，那么对应的 double 数组中的那个字节就设为-1。第 4 行定义了用于提取位信息的 flag 变量。第 6 行提取对应数据位上的数据，第 11 行重新设置 flag，准备提取下一位数据。

对一个字符串中的每一个字符运用上述方法，便可以将一个字符串转为一个数组：

```
1 public static double[] str2Doubles(String str) {
2    double[] re = new double[str.length() * 8];
3    for (int i = 0; i <str.length(); i++) {
4          double[] dChar = char2Doubles(str.charAt(i));
5          System.arraycopy(dChar, 0, re, i * 8, 8);
6    }
7    return re;
8 }
```

有了上面两个函数的帮助，就可以准备训练数据了。在这里，准备了 3 个单词让网络记忆，分别是 good，bad 和 test。

```
DataSettrainingSet = new DataSet(32);
trainingSet.addRow(new DataSetRow(str2Doubles("good")));
trainingSet.addRow(new DataSetRow(str2Doubles("bad ")));
trainingSet.addRow(new DataSetRow(str2Doubles("test")));
```

统一将这 3 个单词向 4 字节长度对齐，不足之处用空格补全。因此，每个单词可以使用固定的 32 位表示。对应到训练数据，就是 32 个字节。构造一个 Hopfield 网络，其规模与训练数据集相同：

```
Hopfield myHopfield = new Hopfield(32);
myHopfield.learn(trainingSet);
```

完成训练后，尝试让该网络进行单词回忆，给出 3 个拼写错误的单词：

```
recallWords("jood", myHopfield);
recallWords("bbd ", myHopfield);
recallWords("tast", myHopfield);
```

上述的 recallWords()函数用于单词联想，给出一个错误的拼写，观察网络是否可以回忆起其正确的拼写方式。执行上述代码，输出如下：

```
Input: jood
Output: good
Input: bbd
Output: bad
Input: tast
Output: test
```

可以看到，在这个案例中，全部的单词都被正确地回忆起来了。最后，来看一下较为关键的 recallWords()函数的实现。recallWords()函数的第一个参数为要识别的单词，它可以是一个拼写错误的单词，这个单词定义了 Hopfield 网络的初态。它的第二个参数是目标 Hopfield 网络：

```
01 public static void recallWords(String word, Hopfield myHopfield) {
02   DataSetRow h = new DataSetRow(str2Doubles(word));
03   myHopfield.setInput(h.getInput());
04   double[] networkOutput = null;
```

```
05   double[] preNetworkOutput = null;
06   while (true) {
07         myHopfield.calculate();
08         networkOutput = myHopfield.getOutput();
09         if (preNetworkOutput == null) {
10               preNetworkOutput = networkOutput;
11               continue;
12         }
13         if (Arrays.equals(networkOutput, preNetworkOutput)) {
14               break;
15         }
16         preNetworkOutput = networkOutput;
17   }
18   System.out.println("Input: " + HopfieldWordRemember.doubles2Str(h.getInput()));
19   System.out.println("Output: " + HopfieldWordRemember.doubles2Str(networkOutput));
20 }
```

上述代码第 2 行，将要识别的单词转为 DataSetRow 的格式，作为网络的初态（第 3 行）。第 6～17 行为网络的反馈振荡，如果满足第 13 行的条件，表示网络已经稳定，则跳出循环体。在网络稳定之后，打印出网络的初态（输入数据）和最终的吸引子（识别结果）。

8.6 总结

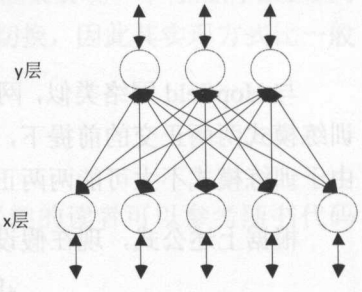
本章详细介绍了离散型 Hopfield 网络的基本原理、实现和典型应用。离散型 Hopfield 网络是一种无监督的自反馈网络，具备一定的自联想能力。在本章中，给出了两个典型的自联想的案例，一是有关污染字符的回忆，二是对拼写错误单词的纠正。可以看到，Hopfield 了网络在这两个案例中，都有着不错的表现。如果需要使用 Hopfield 网络记忆更多的模式，那么就需要构造一个规模更大的网络。为保证网络的训练质量，根据网络的存储容量的基本原理，Hopfield 网络的神经元数量可以设置为需要记忆模式数量的 10 倍以上。

第9章　BAM双向联想记忆网络

上一章介绍了Hopfield网络，该网络具备一定的联想记忆功能。但是，Hopfield的联想能力仅仅局限于自联想。在本章中，将详细介绍另外一种联想网络——双向联想记忆网络Bidirectional Associative Memory，简称BAM网络。与Hopfield网络类似，BAM网络也有离散型和连续型等多种形式，本章主要介绍离散型BAM网络。

9.1 BAM网络的结构与原理

BAM网络是一种双向双层网络。所谓双层，表示BAM有两层结构，双向则表示在这两层结构中，每一层均可作为输入层，也可以作为输出层。图9-1所示显示了这种网络结构。可以看到，该网络由两层组成，x层5个神经元，y层包含3个神经元。层内的神经元没有连接，层与层之间神经元两两连接。



▲图9-1　BAM网络结构

网络在工作时，首先由一端输入数据，比如数据首先从x层传入。根据连接权值，数据达到y层，在y层，数据通过y层神经元的传输函数得到y层的输出。之后，改变x层和y层的角色。将原本作为输入的x层作为输出层，原本作为输出的y层作为输入层。将上一次y层的输出作为y层的输入，通过连接权值传回到x层。在x层，通过x层的传输函数，在x层进行输出。然后，再次改变x层和y层的角色，将x层作为输入，y层作为输出，以此类推，不断迭代，一直到达稳定，即两侧的神经元在运行过程中不再发生变化。

可以看到，从整体形式上来说，BAM 网络和 Hopfeild 网络有异曲同工之妙。Hopfield 网络在单层中进行自迭代，而 BAM 则在两层之间进行迭代。它们最终都要求网络达到稳定状态。对于 BAM 网络来说，当它达到稳态时，不仅可以实现双向联想，而且能对输入侧的模式进行修复。

9.2 BAM 网络的学习算法

对于离线 BAM 网络，一般两侧均选取 *Sgn* 函数作为传输函数。当网络需要存储一对模式（$\boldsymbol{X}^1$，$\boldsymbol{Y}^1$）时，若要使网络达到稳定状态，应该满足如下条件：

$$Sgn(\boldsymbol{W}\boldsymbol{X}^1)=\boldsymbol{Y}^1$$
$$Sgn(\boldsymbol{W}^{\mathrm{T}}\boldsymbol{Y}^1)=\boldsymbol{X}^1$$

其中 $\boldsymbol{W}$ 为权值矩阵。一种符合条件的 $\boldsymbol{W}$ 可以通过以下公式计算，即 $\boldsymbol{W}$ 为向量 $\boldsymbol{X}$ 和 $\boldsymbol{Y}$ 的外积：

$$\boldsymbol{W}=\boldsymbol{Y}^1\boldsymbol{X}^{1\mathrm{T}}$$
$$\boldsymbol{W}^T=\boldsymbol{X}^1\boldsymbol{Y}^{1\mathrm{T}}$$

如果要存储 N 对模式，则可以使用以下公式，对 N 对模式的外积求和：

$$\boldsymbol{W}=\sum_{n=1}^{N}\boldsymbol{Y}^n(\boldsymbol{X}^n)^{\mathrm{T}}$$
$$\boldsymbol{W}^{\mathrm{T}}=\sum_{n=1}^{N}\boldsymbol{X}^n(\boldsymbol{Y}^n)^{\mathrm{T}}$$

与 Hopfield 网络类似，网络权值都是使用训练的模式样本数据求得，并且只有在这些训练模式两两正交的前提下，才能保证网络进行正确的联想。也就是说，在通常使用中，由于训练模式不太可能两两正交，因此网络联想实际上是可能出错的。

根据上述公式，现在假设要设计一个 BAM 网络用于记忆下面两对模式：

$$\boldsymbol{A}^1=(1,0,1,0,1,0),\boldsymbol{B}^1=(1,1,0,0)$$
$$\boldsymbol{A}^2=(1,1,1,0,0,0),\boldsymbol{B}^2=(1,0,1,0)$$

首先，需要将它们映射到 *Sgn* 函数的值域中：

$$\boldsymbol{X}^1=(1,-1,1,-1,1,-1),\boldsymbol{Y}^1=(1,1,-1,-1)$$
$$\boldsymbol{X}^2=(1,1,1,-1,-1,-1),\boldsymbol{Y}^2=(1,-1,1,-1)$$

接着，使用公式 $\boldsymbol{W}^{\mathrm{T}}=\sum_{n=1}^{N}\boldsymbol{X}^{n}(\boldsymbol{Y}^{n})^{\mathrm{T}}$ 计算 $\boldsymbol{W}^{\mathrm{T}}$ 得到：

$$\begin{pmatrix} 2 & 0 & 0 & -2 \\ 0 & -2 & 2 & 0 \\ 2 & 0 & 0 & -2 \\ -2 & 0 & 0 & 2 \\ 0 & 2 & -2 & 0 \\ -2 & 0 & 0 & 2 \end{pmatrix}$$

现在，让我们来验算一下吧！给出 $\boldsymbol{X}^{1}=(1,-1,1,-1,1,-1)$，计算 $\boldsymbol{X}^{1}$ 与 $\boldsymbol{W}^{\mathrm{T}}$ 的乘积，并运用于 *Sgn* 函数，很容易验证，正好得到了 $\boldsymbol{Y}^{1}$。

在 Hopfield 网络中，我们已经介绍了，网络的容量，也就是网络可以记忆的模式数量，取决于网络的规模。对于 BAM 网络来说也是类似的，在最理想情况下，BAM 网络可以存储和回忆的极限是由神经元数据量较少的那层决定的。比如在本例中，两层包含的输入规模分别是 6 个和 4 个。那么从理论上来说，理想情况下，网络最多能记忆的模式数量就是 4 个。

9.3 使用 Java 实现 BAM 网络

基于上述 BAM 网络的基本原理，本节将使用 Neuroph 框架实现一个标准的 BAM 网络。由于 BAM 网络固有的特点，即输入层和输出层会相互切换，因此其实现方式比一般的神经元网络略微复杂。

9.3.1 BAM 网络的静态结构

首先，BAM 网络依然从 NeuralNetwork 类继承（有兴趣的读者可以参考随书代码中的 BAM 类）：

```
public class BAM extends NeuralNetwork<LearningRule> {
```

与 Hopfield 网络类似，BAM 网络的每一个神经元都需要同时作为输出和输入神经元。所以依然会用 InputOutputNeuron 作为 BAM 网络的神经元类型：

```
1 public BAM(int inputNeuronsCount, int outputNeuronsCount) {
2     // init neuron settings for BAM network
```

```
3    NeuronProperties neuronProperties = new NeuronProperties();
4    neuronProperties.setProperty("neuronType", InputOutputNeuron.class);
5    neuronProperties.setProperty("bias", new Double(0));
6    neuronProperties.setProperty("transferFunction", TransferFunctionType.SGN);
7    this.createNetwork(inputNeuronsCount, outputNeuronsCount, neuronProperties);
8 }
```

上述代码设置神经元的偏置为 0，传输函数为符号函数 *Sgn*。

其中，createNetwork()函数的实现如下：

```
01 private void createNetwork(int inputNeuronsCount, int outputNeuronsCount,
02          NeuronProperties neuronProperties) {
03   this.setNetworkType(NeuralNetworkType.BAM);
04   Layer inputLayer = LayerFactory.createLayer(inputNeuronsCount, neuronProperties);
05   this.addLayer(inputLayer);
06   Layer outputLayer = LayerFactory.createLayer(outputNeuronsCount, neuronProperties);
07   this.addLayer(outputLayer);
08   ConnectionFactory.fullConnect(inputLayer, outputLayer, 0);
09   ConnectionFactory.fullConnect(outputLayer, inputLayer, 0);
10   NeuralNetworkFactory.setDefaultIO(this);
11   this.setLearningRule(new BAMLearningRule());
12 }
```

上述代码第 3～7 行构造了 BAM 网络的两个层，第 8～9 行设置了两层之间的连接。与 BP 神经网络不同的是，在这里连接是双向的。即，既有从输出层到输入层的连接，也有输入层到输出层的连接。第 11 行，设置了网络的学习算法为 BAMLearningRule。

9.3.2　BAM 网络学习算法

BAMLearningRule 根据外积和法进行实现，即对训练数据执行外积求和，将结果作为权值：

```
01 @Override
02 public void learn(DataSet trainingSet) {
03   int M = trainingSet.size();
04   for (int i = 0; i < M; i++) {
05         DataSetRow row = trainingSet.getRowAt(i);
06         learnRow(row);
07   }
08 }
09
10 public void learnRow(DataSetRow row) {
```

```
11   for (int i = 0; i < row.getInput().length; i++) {
12         for (int j = 0; j < row.getDesiredOutput().length; j++) {
13               Neuron ini = neuralNetwork.getLayerAt(0).getNeuronAt(i);
14               Neuron outj = neuralNetwork.getLayerAt(1).getNeuronAt(j);
15               outj.getConnectionFrom(ini).getWeight().value +=
16                     row.getInput()[i] * row.getDesiredOutput()[j];
17               ini.getConnectionFrom(outj).getWeight().value =
18                     outj.getConnectionFrom(ini).getWeight().value;
19         }
20   }
21 }
```

上述代码是 BAMLearningRule 算法的核心实现。第 10 行的 learnRow()函数将指导 BAM 网络学习单个样本。根据外积和法的学习规则，$\boldsymbol{W}$ 的每一个分量为样本对应分量的乘积求和（第 15～16 行），同时，由于网络的输入输出层可以互换，切换后，对应的权值矩阵 $\boldsymbol{W}$ 也将变为 $\boldsymbol{W}^{\mathrm{T}}$。因此，有第 17～18 行。对于 M 条样本数据，依次执行 learnRow()即可完成对全部样本的学习。

9.3.3 BAM 网络的运行

有了网络结构和学习算法，接下来就需要实现网络的运行过程。在这里说的网络运行过程，指的是在学习过程结束后，给定一个输入，网络通过振荡迭代达到稳态的过程。完整的振荡过程如下：

```
01 public void calculate(DataSetRow row) {
02   boolean stable1 = true, stable2 = true;
03   do {
04        stable1 = propagateLayer(row);
05        switchInputOutput();
06        switchInputOutputData(row);
07        stable2 = propagateLayer(row);
08        switchInputOutput();
09        switchInputOutputData(row);
10   } while (!stable1 || !stable2);
11   fireNetworkEvent(new NeuralNetworkCalculatedEvent(this));
12 }
```

其中，输入参数 row 表示要识别的一条样本。变量 stable1 和 stable2 分别表示 BAM 网络的两层的稳定性，只有当两层都稳定后，calculate()函数的执行才会结束。第 4 行代码将输入数据通过输入层传输到输出层，并返回输出层的稳定性。接着，在第 5～6 行，

切换输入和输出层的角色，同时训练数据的输入输出也进行切换。切换输入输出后，在第 7 行将数据反向传递回来。最后，在第 8～9 行再次切换输入输出，为下一次振荡进行准备。第 10 行为循环结束的条件，即两层节点均达到稳定。

一次数据传输过程的具体实现如下：

```
01 public boolean propagateLayer(DataSetRow row) {
02   this.setInput(row.getInput());
03   for (Layer layer : this.getLayers()) {
04         layer.calculate();
05   }
06   boolean stable = true;
07   double[] output = this.getOutput();
08   for (int i = 0; i < output.length; i++) {
09         if (output[i] != row.getDesiredOutput()[i]) {
10               row.getDesiredOutput()[i] = output[i];
11               stable = false;
12         }
13   }
14   return stable;
15 }
```

上述代码中，第 2 行设置网络一侧的输入，第 3～5 行进行数据运算，其结果是在网络另一侧产生一个对应的输出。第 7 行得到这个输出，存储在 output 变量中。第 8～13 行将网络的实际输出和期望输出进行比较，如果不相同，则将实际输出作为下一次的期望输出。因此，这里的期望输出也起到了保存上一次输出的作用，方便判断网络是否达到稳态，如果后续期望输出和实际输出完全相同，则表示该层已经稳定。此外，第 10 行赋值语句还包含一个非常重要的隐含逻辑：网络的当前输出，将立即作为后一次振荡的输入。在 propagateLayer()函数执行后，紧接着就是网络结构和输入输出的切换。下面代码显示了输入输出数据的切换：

```
1 public void switchInputOutputData(DataSetRow row) {
2      double[] t = row.getInput();
3      row.setInput(row.getDesiredOutput());
4      row.setDesiredOutput(t);
5 }
```

可以看到，原先的输入成为输出，而原来的输出变成了后一次的输入。这就是为什么要将网络输出保存在 desiredOutput 中的重要原因。

网络结构的切换也类似这样的逻辑。网络在执行过程中，需要重新定义输出层和输出层：

```
1 public void switchInputOutput() {
2    Neuron[] tmp = this.getInputNeurons();
3    this.setInputNeurons(this.getOutputNeurons());
4    this.setOutputNeurons(tmp);
5    Layer[] layers = getLayers();
6    Layer t = layers[0];
7    layers[0] = layers[1];
8    layers[1] = t;
9 }
```

上述代码第 2～4 行互换了输入输出神经元，第 5～8 行互换了输入输出层。

至此，一个完整的 BAM 网络就完成了。下面，就让我们通过一个小例子，来体验一下 BAM 网络的特殊能力吧！

9.4 BAM 网络的应用

BAM 网络是一个双向联想网络。它拥有非常广泛的使用场景，可以建立两个事物之间的对应关系。例如，我们建立了字符 A 和 B 的对应关系，给出 A 则应该输出 B；给出 B，则应该输出 A。这有点像数据结构中的映射，但比映射更为奇妙的是，对于映射来说，为了得到 B，你必须给出一个完全正确的 A。如果你给出的 A 对象有一点点的污损或者噪点，映射就无法找到对应的 B。而这正是 BAM 网络比映射更为奇妙的地方。对 BAM 网络来说，即使给出的 A 带有一点噪点，它依然有可能找到与之对应的 B。同时，还会将 A 的噪声清理，得到一个完整的 A 对象。

9.4.1 场景描述——人名与电话

BAM 网络应用的一个典型例子，就是人名和电话号码的记忆。在我们的手机通信录里，通常会保存一些电话号码。这些电话号码总是会与一个人名相对应。比如一个名叫 TINA 的联系人，他的电话是 6843726。那么，TINA 和 6843726 之间就有着很强的映射关系。如果这是一个你经常访问的家人，如父母，你应该很容易地就能从 TINA 联想到 6843726。同时，如果你只看到 6843726，也一定会觉得很熟悉，很自然地会想到这个号码的主人 TINA。更进一步地，如果有人把 TINA 的名字写错成了 TINE，与 TINA 很像，

如果此时你也正在找 TINA 的联系方式，你是不是也会通过 TINE 直接联想到 TINA 以及 6843726 呢？同理，如果给出了一个错误的号码 6843725，你自然也会想到你的一个家人的电话恰好是 6843726，而这个人叫作 TINA。

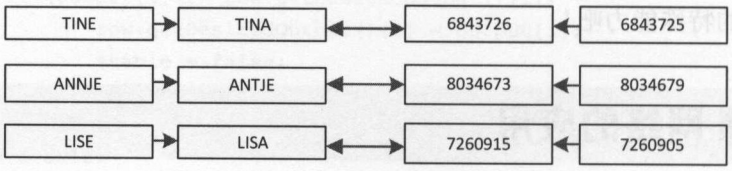
这里给出 3 个联系人以及他们的联系方式：

TINA　6843726

ANTJE 8034673

LISA　7260915

现在，就可以构造一个 BAM 网络，让它记住上述人名和电话。希望达到的效果如图 9-2 所示。



▲图 9-2　BAM 期望执行效果

对于输入正确的人名和电话，网络需要给出正确的输出。如果在输入数据上进行一点扰动，如 TINA 拼写成 TINE，8034673 误写成 8034679，网络不仅需要把数据还原，而且需要找到对应的映射数据。

9.4.2　数据编码设计

为了让程序更好地完成任务，我们需要设计一套合理的编码体系。毕竟，对于 BAM 网络来说，它只能识别 1 和−1 的输入。如何将一个字符串编码成由 1 和−1 组成的数组，是首先需要解决的问题。

在本例中，为简单起见，对于人名统一不区分大小写，并且不支持特殊符号。这样，我们就大大缩小了可用字符的范围，更有利于精简编码。一般来说，一个可见字符通过 ASCII 编码后，为 8 位。假设一个字符串包含 5 个字符，那么内部就需要 40 位来存储这个字符串。由于神经元只能接受或输出 1 和−1 的数据，这就意味每一位都需要对应一个神经元，换言之，输入一个长度为 5 的字符串，就需要多达 40 个神经元。我们知道，当数据在 BAM 网络中传递时，神经元数量越多，计算成本就越高，对于矩阵相乘来说，其时间复杂度可以达到 $O(n^3)$。所以，如果有可能，减少神经元的数量将对 BAM 网络的性

能带来一定的提升。

基于上述原因，在本例中，统一将一个字符编码成 6 位，即使用 6 个神经元就可以表示一个字符。这样，对于长度为 5 的字符串，只需要 30 个神经元就能很好地处理，大大降低了计算成本。

下面一段代码，将一个输入字符串转为一个 double 数组，该返回的 double 数组中，只包含 1 和−1：

```
01 public static final int BITS_PER_CHAR = 6;
02 public static final char FIRST_CHAR = ' ';
03 public static double[]stringToBipolar(String str) {
04   double[] result = new double[str.length() * BITS_PER_CHAR];
05   int currentIndex = 0;
06   for (int i = 0; i < str.length(); i++) {
07         char ch = Character.toUpperCase(str.charAt(i));
08         int idx = ch - FIRST_CHAR;
09
10         int place = 1;
11         for (int j = 0; j < BITS_PER_CHAR; j++) {
12               boolean value = (idx & place) > 0;
13               result[currentIndex++] = value ? 1 : -1;
14               place <<= 1;
15         }
16   }
17   return result;
18
19   }
```

上述代码中，BITS_PER_CHAR 为 6，表示将一个字符压缩到 6 位。FIRST_CHAR 为空格，这是因为空格位于所有可见 ASCII 字符中的第一位。这样空格可以作为一个新的基准，所有的可见字符减去空格的 ASCII 码后，再进行处理，这样就可以将字符压缩到 6 位。这就是上述代码第 8 行的用意。

以字符 Z 为例，Z 的 ASCII 码为 0101 1010，第 7 位上为 1，压缩到 6 位后，就会丢失；而 Z 减去空格后，为 00111010，正好为 6 位，压缩后没有损失。

第 11～19 行读取一个字符中每一位的值，并对应到 result 数组中。第 13 行设置了对应关系，即 1 表示为 1，0 表示为−1。

有了编码操作，必然还需要对应的解码操作。也就是当网络稳定后，我们需要将网络的输出进行解码，还原成可读的字符串。解码操作是编码操作的逆运算：

```
01 public static String bipolalToString(double[] data) {
02   StringBuilder result = new StringBuilder();
03
04   int j, a, p;
05
06   for (int i = 0; i < (data.length / BITS_PER_CHAR); i++) {
07         a = 0;
08         p = 1;
09         for (j = 0; j < BITS_PER_CHAR; j++) {
10               if (data[(i * BITS_PER_CHAR + j)] > 0)
11                     a += p;
12               p <<= 1;
13         }
14         result.append((char) (a + FIRST_CHAR));
15   }
16   return result.toString();
17 }
```

上述代码中，将网络的输出 double 数组逆向转为对应的字符串。它是 stringToBipolar() 函数的反向操作。

9.4.3　具体实现

在正式实现这个功能之前，先进行以下规定。

- 人名的字符串长度不超过 5 个字符；不到 5 个字符，用空格补全到 5 个字符。
- 电话号码不超过 7 个字符。

首先，定义需要记忆的人名和电话号码：

```
public static final String[] NAMES = { "TINA ", "ANTJE", "LISA " };
public static final String[] PHONES = { "6843726", "8034673", "7260915" };

public static final String[] NAMES_BAD = { "TINE ", "ANNJE", "LISE " };
public static final String[] PHONES_BAD = { "6843725", "8034679", "7260905" };
```

上述代码中的 NAMES 和 PHONES 常量定义了需要记忆的人名和电话，其中

NAMES[0]对应 PHONES[0]，即 TINA 的电话为 6843726，以此类推。NAMES_BAD 和 PHONES_BAD 常量定义了一些受损的输入，比如 NAMES[0]对应 NAMES_BAD[0]，表示 TINA 被误写成 TINE 传入 BAM 网络；PHONES[0]对应 PHONES_BAD[0]，表示 6843726 被误写成 6843725 传入网络。

定义初始的网络结构和数据集结构：

```
public static final int IN_CHARS = 5;
public static final int OUT_CHARS = 7;
public static final int BITS_PER_CHAR = 6;
public static final int INPUT_NEURONS = (IN_CHARS * BITS_PER_CHAR);
public static final int OUTPUT_NEURONS = (OUT_CHARS * BITS_PER_CHAR);

BAM logic = new BAM(INPUT_NEURONS, OUTPUT_NEURONS);
DataSet ds = new DataSet(INPUT_NEURONS, OUTPUT_NEURONS);
```

其中 IN_CHARS 表示初始的输入数据人名占用 5 个字符，OUT_CHARS 表示初始的输出电话号码占用 7 个字符，BITS_PER_CHAR 表示每个字符占用 6 个比特；INPUT_NEURONS 表示输入神经元数量，OUTPUT_NEURONS 表示输出神经元数量。根据输入和输出神经元数量，就可以定义 BAM 网络和它的训练数据集。

有了网络和数据集后，就可以开始训练 BAM 网络：

```
1 for (int i = 0; i < NAMES.length; i++) {
2     DataSetRow dr = new DataSetRow();
3     dr.setInput(stringToBipolar(NAMES[i]));
4     dr.setDesiredOutput(stringToBipolar(PHONES[i]));
5     ds.addRow(dr);
6 }
7
8 logic.learn(ds);
```

上述代码第 3～4 行将 NAMES 和 PHONES 转为网络可以接受的数据集格式，第 8 行执行网络的学习过程。

学习过程结束后，网络中的权值也就固定下来了。后续可以给出一条待识别的数据，去运行这个网络：

```
1 public static void runBAM(BAM logic, DataSetRow data) {
2    StringBuilder line = new StringBuilder();
```

```
3     line.append(mappingToString(data));
4     logic.calculate(data);
5     line.append("  |  ");
6     line.append(mappingToString(data));
7     System.out.println(line.toString());
8  }
```

上述代码中定义了函数 runBAM()。这个函数接受两个参数，第一个参数为已经完成学习过程的 BAM 网络，第二个参数为要识别的一条数据。其中，mappingToString()函数将一条数据进行解码，转为人类可读的字符串。第 4 行将数据传入网络进行迭代振荡。第 6 行将网络稳定后的输入进行展示。函数 mappingToString()主要调用解码函数 bipolalToString()处理输入和输出数据：

```
1 public static String mappingToString(DataSetRow row) {
2     StringBuilder result = new StringBuilder();
3     result.append(bipolalToString(row.getInput()));
4     result.append(" -> ");
5     result.append(bipolalToString(row.getDesiredOutput()));
6     return result.toString();
7 }
```

有了 runBAM()函数后，就可以很容易地将任意输入传入网络并获得其稳定的输出。下面一段代码将原始的人名传入网络：

```
1 for (int i = 0; i < NAMES.length; i++) {
2     DataSetRow dr = new DataSetRow();
3     dr.setInput(stringToBipolar(NAMES[i]));
4      dr.setDesiredOutput(randomBiPolar(OUT_CHARS * BITS_PER_CHAR));
5    runBAM(logic, dr);
6 }
```

经过网络运行并达到稳态后，其输出如下：

```
TINA  -> 1HR>]JP  |  TINA  -> 6843726
ANTJE -> :D:3CLF  |  ANTJE -> 8034673
LISA  -> CC\AQZ8  |  LISA  -> 7260915
```

可以看到，网络已经对另外一端的电话号码产生了正确的联想。

尝试输入一些错误的人名：

```
1 for (int i = 0; i < NAMES_BAD.length; i++) {
2     DataSetRow dr = new DataSetRow();
3     dr.setInput(stringToBipolar(NAMES_BAD[i]));
4     dr.setDesiredOutput(randomBiPolar(OUT_CHARS * BITS_PER_CHAR));
5     runBAM(logic, dr);
6 }
```

网络稳定后，其输出如下：

```
TINE  -> "W#;U>   |  TINA  -> 6843726
ANNJE -> "YUM!2&  |  ANTJE -> 8034673
LISE  -> ,0[\%:+  |  LISA  -> 7260915
```

可以看到，网络不仅对另外一端的电话号码进行了联想，而且纠正了输出错误的人名信息。

最后，尝试从电话号码到人名的联想，首先输入正确的电话号码，观察网络是否能够进行正确的联想：

```
1 logic.switchInputOutput();
2 System.out.println();
3 for (int i = 0; i < PHONES.length; i++) {
4     DataSetRow dr = new DataSetRow();
5     dr.setInput(stringToBipolar(PHONES[i]));
6     dr.setDesiredOutput(randomBiPolar(IN_CHARS * BITS_PER_CHAR));
7     runBAM(logic, dr);
8 }
```

特别注意第 1 行代码，因为现在网络将原先的输出层作为输入层，因此必须要切换网络的输入和输出。第 5 行代码将正确的电话作为切换输入输出后网络的输入。将这些输入传入网络并运行，得到结果如下：

```
6843726 -> .:-'9  |  6843726 -> TINA
8034673 -> 69H@&  |  8034673 -> ANTJE
7260915 -> Y."("  |  7260915 -> LISA
```

可以看到，网络通过电话号码已经正确联想到了对应的联系人。

那输入错误的电话号码又将如何呢？

```
1 for (int i = 0; i < PHONES_BAD.length; i++) {
2     DataSetRow dr = new DataSetRow();
3     dr.setInput(stringToBipolar(PHONES_BAD[i]));
```

```
4     dr.setDesiredOutput(randomBiPolar(IN_CHARS * BITS_PER_CHAR));
5     runBAM(logic, dr);
6 }
```

上述代码第 3 行，将错误的电话传递给网络，网络稳定后，输出如下：

```
6843725 -> 7*ZQE  |  6843726 -> TINA
8034679 -> NW:)Z  |  8034673 -> ANTJE
7260905 -> [8"V^  |  7260915 -> LISA
```

不难看出，网络通过错误的号码得到了正确的联系人姓名，并且顺利纠正了错误的号码信息。

在上述实验中，对于每一次网络输入，我们并不关心 DataSetRow 中的 desiredOutput 信息，因为输出信息会由网络振荡迭代自行恢复。所以，在这里统一使用随机函数 randomBiPolar()产生一组随机向量。这也就是网络在运行前，解码后会得到一组乱码的原因。函数 randomBiPolar()的实现如下：

```
public static double[] randomBiPolar(int size) {
     double[] result = new double[size];
     for (int i = 0; i < size; i++) {
          if (Math.random() > 0.5)
               result[i] = -1;
          else
               result[i] = 1;
     }
     return result;
}
```

9.5 总结

本章主要讨论了一种双向联想记忆网络 BAM。与 Hopfield 相比，BAM 似乎具有更加神奇的功能，它不仅可以像映射一样建立数据的对应关系，而且在通过合理的学习后还具备还原原始数据的能力，是一种有着广泛使用场景的神经网络。

第 10 章　竞争学习网络

本书前面已经介绍了两种类型的神经网络学习算法——监督式学习神经网络与非监督式学习网络。监督式学习网络在学习训练时，需要使用大量的打标数据，网络将打标数据作为参考答案，以此判断网络误差，并根据误差调整网络中的权值连接，从而提高网络拟合类似数据的能力，这类网络以 BP 神经网络为代表；非监督式学习网络则不需要打标数据，网络根据数据本身的规律，通过不停的观察，自行揭露数据内在的联系，这类网络以 Hopfield 网络为代表。本章将要介绍的竞争学习网络属于这种非监督式的学习。

10.1 竞争学习的基本原理

竞争学习，顾名思义就是在学习过程中，要求不同神经元之间进行竞争，胜利的神经元才能激活，失败的神经元则必须抑制。并且，只有胜利的神经元才有资格更新权重，失败的神经元不改变其权重。

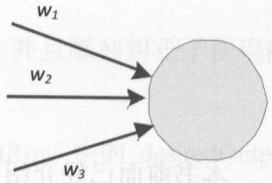
这种思想来源于对生物神经系统的研究。在生物学上，神经系统接受一个外部刺激，那么可能只有某一些神经细胞会对此作出强烈反应而被激活，在此神经细胞周围的一些神经细胞其表现特性也会比较接近这个激活细胞，而远离激活细胞的神经细胞则表示出截然不同的姿态，即被抑制。比如，生物视网膜中，有许多特定的细胞对特定的图像特别敏感。当视网膜中若干个接受单元同时受到刺激，就会使大脑中的特定区域激活，从而使得大脑分辨出特定的图像。并且，越是接近的模式输入所激活的神经细胞也越接近，这样在视觉上就能有比较强的容错能力。

这种将固定类似模式对应到特定神经细胞上的能力并不是天生的，而是后天习得的。这正是一种非监督式学习。通过大量的训练样本，逐步形成特定模式与神经细胞间的对应

关系。

10.1.1　向量的相似性

在竞争学习中，一个非常重要的规则就是：如何判别某一个神经元获胜。我们已经知道，任何一个输入，其数学本质就是一个向量。而一个神经元输入连接上的权重也构成一个向量，如图 10-1 所示，w_1，w_2，w_3 就构成一个向量。



▲图 10-1　神经元的权重向量

我们只需要比较神经元的输入向量和输入权重向量的关系，就可以确定哪一个神经元更为接近输入数据。最接近输入向量的神经元即为获胜神经元，可以被激活，并且更新其权重值。通常，可以使用两种方式来判断向量的相似性。

（1）欧式距离。

欧式距离就是通常所说的距离，也就是比较两个向量在坐标轴上的距离。距离越近，表示两个向量越接近，反之，则表示两个向量越远。对于给定输入模式 X，只有权重向量 W 距离 X 最近的那个神经元区域被激活。W 和 X 的距离定义如下(假设只有两个分量)：

$$\sqrt{(w_1-x_1)^2+(w_2-x_2)^2}$$

比如，向量[1 2]和向量[3 4]。它们的欧式距离为：

$$\sqrt{(1-3)^2+(2-4)^2}=2\sqrt{2}$$

（2）余弦相似度。

除距离外，另一种判别向量相似度的方式是测量向量的夹角。向量夹角越小，表示两个向量越相似。如果向量夹角为φ，当φ越小时，cosφ就越大，因此可以通过 cosφ来判断向量的相似度，即 cosφ越大，表示两个向量越相似。cosφ的计算公式如下：

$$\cos\varphi=\frac{X^{\mathrm{T}}X}{\|W\|\|X\|}$$

需要注意的是，如果使用余弦相似度进行判别，那么通常需要将向量先进行归一化操作。即在不改变向量方向的前提下，将向量拉升或者压缩成单位向量，向量的归一化公式如下：

$$\hat{X}=\frac{X}{\|X\|}$$

即将向量中的每一个分量除以向量的长度。比如，对于向量[1 2 3 4]，其长度为 $\sqrt{1^2+2^2+3^2+4^2}=5.477226$。将每个分量除以 5.477226 后，得到归一化后的向量：[0.1825742 0.3651484 0.5477226 0.7302967]。

10.1.2 竞争学习规则

有了向量的相似性度量，我们就可以开始构建竞争学习算法。竞争学习算法主要可以分为 3 步。

（1）向量归一化。

向量归一化，可以看作数据的预处理。它将一个向量拉升或者压缩为另外一个方向不变的单位向量。在竞争学习中，无论是输入模式向量还是神经元的权重向量，都需要归一化。

（2）通过竞争得到获胜神经元。

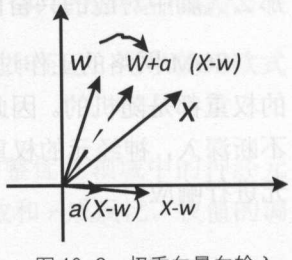
在这一步中，通过度量输入向量和各个神经元权重向量的相似性，将与输入向量最为接近的神经元作为获胜神经元。相似性的度量方法可以是欧氏距离，也可以是余弦夹角。如果使用欧式距离，那么第一步的向量归一化可以省略。

（3）调整获胜神经元的权重向量 W。

假设神经元 j 取得胜利，那么 w_j 则调整为：

$$w_j = w_j + \alpha(\hat{X} - \hat{w}_j)$$

其中，w_j 为神经元 j 的权重向量，α 为步长，即学习率，通常取 0 到 1 之间的浮点数。$\hat{X}$ 为归一化后的输入向量。除神经元 j 以外，其余神经元的权重向量均得不到调整。从这个公式中也不难看出，在进行调整后，原始的 w_j 向量事实上是向着输入模式的方向进行了偏移，如图 10-2 所示。



▲图 10-2 权重向量向输入向量 X 进行偏移

这样，权重向量就更加接近输入的向量，当后续有类似的样本向量进入网络时，该神经元获胜的概率也就大大增加。最终，将在整个系统中，建立特定神经元对应特定模式向量的关系。

由于网络在权值向量调整后，权值向量将不再是单位向量，所以在调整后，应该再次对权值向量进行归一化。在第 3 步完成后，通常网络需要进行若干次迭代，也就是网络的学习过程会再次回到步骤 1，并不停反复进行。

迭代的终止条件可以有多种，常用的策略有：

（1）迭代次数控制，即网络对训练数据进行固定次数的迭代训练。到达一定迭代次数后，立即终止学习。

（2）根据学习率α进行衰减。在网络学习初期，α可以设置比较大，比如 0.9。随着网络的学习，网络权重应该逐步稳定，网络也应该趋于稳定。此时，可以根据迭代次数，不断衰减α，直到 0 为止。当α衰减到 0，则停止学习。

10.2 自组织映射网络 SOM 的原理

自组织映射网络（Self-Organizing Feature Map，简称 SOM），是在 1981 年由芬兰赫尔辛基大学 Kohonen 教授提出的。他认为，一个神经网络接受外界输入后，将会分为不同的对应区域，也就是不同的输入模式会激活网络中不同的神经元。因此，对于某一个网络来说，给定不同的输入模式，将会有不同的响应特征。类似的输入模式也将会有类似的响应特征或者响应区域。并且，这个映射关系是后天训练自动习得的。这个网络的特征与人脑的学习方式非常类似。

10.2.1　SOM 网络的生物学意义

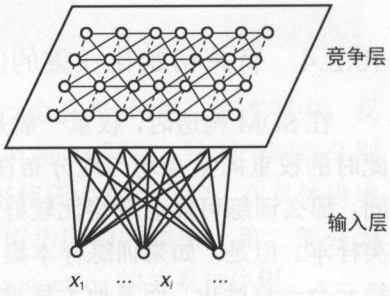
从生物学角度上说，人脑的神经元排列是有序的。大脑中不同的区块会处理不同的感官刺激。比如，生物视网膜上许多特定的细胞对特定的图形或颜色比较敏感，而这些细胞能使得大脑中特定神经元兴奋，从而使得大脑得以对输入信号进行判断。输入模式越接近，那么大脑中对应的兴奋区域也越接近。这些都是人在出生后自然习得的。

SOM 网络的工作过程也非常类似。对于一个 SOM 网络来说，在网络初始化时，网络的权重都是随机的。因此，哪个神经元会对哪些模式做出响应是不确定的。但随着学习的不断深入，神经元的权重会不断调整，最终给定固定的输入模式，就会有特定区域的神经元进行响应。

10.2.2　SOM 网络的结构

SOM 网络也是分层网络。但 SOM 通常只有两层，第一层为输入层，对应输入模式，这与 BP 神经网络一样。第二层为输出层，也就是竞争层。与 BP 神经网络的不同的是，在 SOM 中，竞争层可以有不同的形态，可以是线性的，也可以是平面的，甚至是三维网

格的。一般来说，平面的竞争层较为常用，如图 10-3 所示。



▲图 10-3 SOM 网络结构

可以看到，输入层是一维的，对应输入模式，竞争层是一个二维平面，并且竞争层中的每一个神经元都与输入层中的每一个神经元两两连接。给定一个输入模式，在竞争层会对应一个获胜神经元。以获胜神经元为中心，给定一个领域 R，则称这个区域为优胜领域。在优胜领域中的所有神经元都有权利调整其权重，假设优胜领域中的神经元与获胜神经元的距离为 r，那么神经元的学习率α与 r 成反比，即神经元距离获胜点越远，则学习效果越不明显。一种简单的调整方法为：

$$\alpha = \alpha/(\gamma + 1)$$

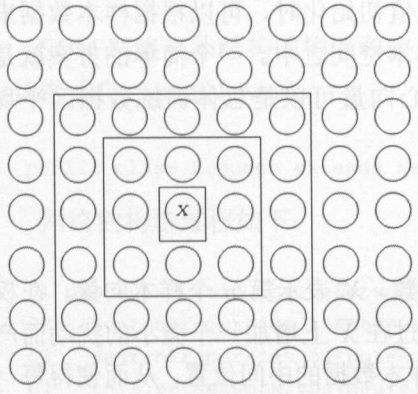
当调整神经元为获胜神经元时，r 为 0。当调整神经元为获胜神经元的邻居时，r 为 1。以此类推。

随着训练的进行，SOM 网络应该趋于稳定，领域 R 也应该不断缩小。

10.2.3 SOM 网络的运行原理

结合竞争学习算法以及 SOM 网络的结构，可以得出一个 SOM 网络的学习规则。该学习规则是由 Kohonen 教授提出的，因此称为 Kohonen 算法，其基本步骤如下。

（1）首先网络需要初始化，权向量应该是较小的随机数，将权向量归一化。定义初始领域 R 以及学习率α。



▲图 10-4 不断缩小的领域半径

（2）将训练模式归一化，并作为网络的输入。

（3）在竞争层选取获胜神经元。可以使用欧式距离，也可以使用余弦夹角。

（4）定义优胜领域，调整优胜领域中的神经元权重。α应该同时与迭代次数和 r 成反比。权值的调整公式：

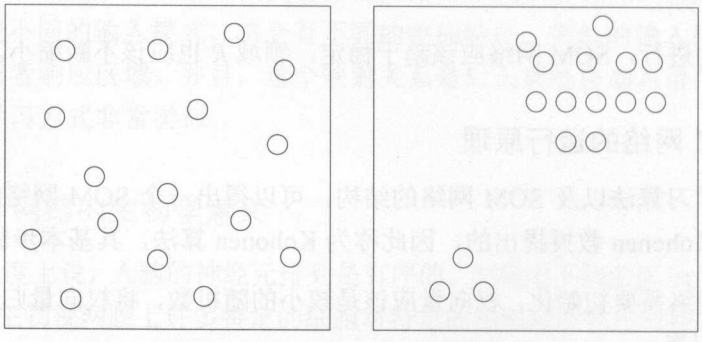
$$w_j = w_j + \alpha(\hat{X} - \hat{w}_j)$$

领域半径 R 也将随着训练的进行而缩小，如图 10-4 所示。

（5）达到迭代次数或者α缩减为 0，则结束学习，否则重复第 1～4 步。

10.2.4　有关初始化权重的问题

在 SOM 构造时，权重一般是随机设定的，通常会设置为一些比较小的随机数。因此，此时的权重向量会均匀地分布在整个向量空间。如果训练样本也恰好均匀分布在整个空间，那么训练可能会取得比较好的效果，最终所有神经元都有机会获胜，并且映射到某一类样本。但是，如果训练样本集中于某一块区域，那么将导致在训练过程中，只有少数神经元会一直胜出，而其他大量神经元始终得不到调整权值的机会。继而，大量的样本都会被映射到同一个或者少数几个神经元，那么训练效果就很差。图 10-5 所示的左图显示了一种比较均匀的情况，对于给定样本会分别激活各个位置上的神经元，这种情况是比较理想的。而右图则表示多个样本对应到了某一个或者少数神经元上，大部分神经元得不到训练的机会，导致大量样本被集中归为一类，这种情况是应该尽量避免的。



▲图 10-5　SOM 神经元的活跃分布

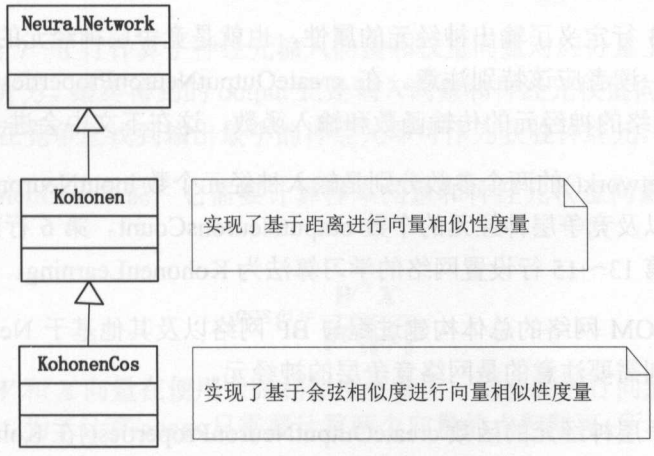
为了尽可能使每一个神经元都有机会获胜，在权值初始化时，可以根据样本数据来初始化权值向量，确保权值向量较为均匀地混杂在样本空间之中。一个简单的方案就是让初始权值围绕在样本的中心向量周围。所谓的中心向量可以是总体训练数据的平均值，即：

$$\overline{X}=\frac{1}{N}\sum_{n=1}^{N}X^n$$

上式中，$\overline{X}$ 为训练数据的中心向量，N 为样本总数，X^n 表示第 n 个样本向量。在权值初始化时，竞争层每一个神经元的权值向量可以通过在 $\overline{X}$ 上增加一个微小的扰动而产生。这样，很大程度上可以保证初始化的权值都落在样本数据的中间位置，从而使得每一个神经元都有机会获胜，进而达到比较好的训练效果。

10.3 SOM 网络的 Java 实现

在了解了基本的原理后，就可以使用 Neuroph 来实现一个 SOM 网络。在本节中，我们将同时实现使用距离和余弦相似度进行相似性度量的两个 SOM 网络。这两个网络分别命名为 Kohonen 和 KohonenCos，有兴趣的读者可以在随书代码中找到它们。在具体讲述实现过程时，我们主要分为两个部分，第一部分为网络拓扑结构的构造，第二部分为 Kohonen 算法的实现。图 10-6 所示显示了 Kohonen 和 KohonenCos 的关系与作用。



▲图 10-6 SOM 网络的类图

10.3.1 SOM 网络拓扑结构的实现

SOM 网络拓扑结构的构造和定义与 BP 神经网络等非常类似。显然，Kohonen 网络也应该继承自 NeuralNetwork：

```
public class Kohonen extends NeuralNetwork
```

网络的构造过程如下：

```
01 protected void createNetwork(int inputNeuronsCount, int outputNeuronsCount) {
02   NeuronProperties inputNeuronProperties = new NeuronProperties();
03   NeuronProperties outputNeuronProperties = createOutputNeuronProperties();
04   this.setNetworkType(NeuralNetworkType.KOHONEN);
05
```

```
06    Layer inLayer = LayerFactory.createLayer(inputNeuronsCount, inputNeuronProperties);
07    this.addLayer(inLayer);
08    Layer mapLayer = LayerFactory.createLayer(outputNeuronsCount, outputNeuron
      Properties);
09    this.addLayer(mapLayer);
10    ConnectionFactory.fullConnect(inLayer, mapLayer);
11    NeuralNetworkFactory.setDefaultIO(this);
12
13    KohonenLearning kl = new KohonenLearning();
14    kl.setIterations(100, 50);
15    this.setLearningRule(kl);
16 }
```

上述代码第 3 行定义了输出神经元的属性，也就是竞争层神经元的属性。这是 SOM 与众不同的地方，读者应该特别注意。在 createOutputNeuronProperties()函数中，定义了用于组成 SOM 网络的神经元的传输函数和输入函数，这在下文中会进一步讲解。

函数 createNetwork()的两个参数分别是输入神经元个数 inputNeuronsCount（也就是输入向量的长度）以及竞争层神经元的个数 outputNeuronsCount。第 6 行建立输入层，第 8 行建立竞争层，第 13～15 行设置网络的学习算法为 KohonenLearning。

不难看出，SOM 网络的总体构建过程与 BP 网络以及其他基于 Neuroph 框架的网络非常相似。但特别需要注意的是网络竞争层的神经元。

定义网络竞争层神经元的函数 createOutputNeuronProperties()在 Kohonen 中如下：

```
protected NeuronProperties createOutputNeuronProperties() {
    return new NeuronProperties(Neuron.class, Difference.class, Linear.class);
}
```

在 KohonenCos 中则如下：

```
protected NeuronProperties createOutputNeuronProperties() {
    return new NeuronProperties(Neuron.class, DotProduct.class, ArcCos.class);
}
```

可以看到，对于基于距离的度量，神经元的输入函数为 Difference，传输函数为 Linear。也就是，竞争层神经元的输出就是输入，而输入则由 Difference 计算得来。这与 BP 网络中的加权求和输入函数 WeightedSum 是完全不同的。

Difference 用于计算输入模式向量与神经元权值向量之间的距离，其核心代码如下：

```
01 public double getOutput(Connection[] inputConnections) {
02   double output = 0d;
03      double sum = 0d;
04   for(Connection connection : inputConnections) {
05        Neuron neuron = connection.getFromNeuron();
06        Weight weight = connection.getWeight();
07        double diff = neuron.getOutput() - weight.getValue();
08        sum += diff * diff;
09   }
10   output = Math.sqrt(sum);
11   return output;
12 }
```

不难看出，第 7～8 行计算了神经元输入向量和权重向量对应分量上的差值的平方。第 10 行将平方和开方，最终得到的 output 正是输入向量和神经元权值向量的距离。因此，在训练时，只需要在竞争层找到输出最小的神经元即可作为获胜神经元，并将其激活更新。

而对于 KohonenCos 来说，它需要计算样本向量和神经元权重向量的余弦值，根据公式

$$\cos\varphi = \frac{\boldsymbol{W}^{\mathrm{T}}\boldsymbol{X}}{\|\boldsymbol{W}\|\|\boldsymbol{X}\|}$$

不难看到，由于 $\boldsymbol{W}$ 和 $\boldsymbol{X}$ 向量在使用余弦相似度度量前，均会先进行向量归一化，故分母 $\|\boldsymbol{W}\|\|\boldsymbol{X}\|$ 为 1。因此，为了计算 $\cos\varphi$，只需要计算两个向量的点积即可。所以，在 KohonenCos 中，使用的输入函数为 DotProduct。其实现如下：

```
1 public double getOutput(Connection[] inputConnections) {
2    double sum = 0d;
3    for (Connection connection : inputConnections) {
4         Neuron neuron = connection.getFromNeuron();
5         Weight weight = connection.getWeight();
6         sum +=  neuron.getOutput() * weight.getValue();
7    }
8    return sum;
9 }
```

上述代码第 6 行，将输入向量和权重向量对应分量相乘并求和，最终得到两个向量的点积。与距离不同，在距离度量时，输出最小的神经元将判断为获胜神经元。但此处，向量相似性与余弦值成反比，所以如果将余弦值直接作为神经元输出，就会要求系统使用输出最大的神经元作为获胜神经元。这样 KohonenCos 和 Kohonen 的算法就很难统一。为了

保证 KohonenCos 和 Kohonen 算法的统一性，这里将 KohonenCos 竞争层的输出函数定义为反余弦函数，即将最终的向量夹角计算出来。这里使用 ArcCos 作为竞争层输出函数，其实现如下：

```
public double getOutput(double net) {
    return Math.acos(net);
}
```

这样，KohonenCos 的竞争层实际将输出两个向量的夹角，在进行相似度判定时，与向量的距离一样，都与相似性成正比，因此可以使用同一套算法框架训练网络。

10.3.2　SOM 网络的初始权值设置

在前面中已经提到，为了保证网络的训练效果，SOM 网络在权值初始化时，需要参考训练数据样本。因此，在网络静态结构创建完成之后，学习算法执行之前，还需要根据训练数据重新调整竞争层神经元的权值向量。所以在 Kohonen 中，重写 learn()函数。在正式训练之前，首先调整网络的初始权值：

```
1 @Override
2 public void learn(DataSet trainingSet) {
3    preLearn(trainingSet);
4    super.learn(trainingSet);
5 }
6
7 public void preLearn(DataSet trainingSet) {
8    randomizeWeights(new CenterVectorWeight(trainingSet));
9 }
```

上述代码重写了 learn()函数，首先执行 preLearn()函数，进行网络初始权值的调整。其中 CenterVectorWeight 类计算训练数据 trainingSet 的中心向量（也就是均值向量），并在中心向量上增加一个微小扰动，作为神经元的权值。

1. 基于中心向量的随机化 CenterVectorWeight

CenterVectorWeight 核心实现如下：

```
01 @Override
02 public void randomize(NeuralNetwork neuralNetwork) {
03     for (Neuron neuron : neuralNetwork.getOutputNeurons()) {
04         setWeightVector(neuron);
```

```
05      }
06  }
07
08  private void setWeightVector(Neuron neuron) {
09      Connection[] inputConnections=neuron.getInputConnections();
10      for (int i=0;i<inputConnections.length;i++) {
11          double newWeight=centerVector[i]+Math.random()-0.5;
12          inputConnections[i].getWeight().setValue(newWeight);
13      }
14  }
```

上述代码中，第 2 行 randomize()函数执行对网络权值的随机调整。在具体设置给定神经元权值时，使用 setWeightVector()函数。在第 11 行，对中心向量 centerVector 进行±0.5 的扰动，将随机扰动后的数据作为神经元权重（第 12 行）。

中心向量 centerVector 在 CenterVectorWeight 的构造函数中计算得出：

```
01 public class CenterVectorWeight extends WeightsRandomizer {
02   private double[] centerVector=null;
03   public CenterVectorWeight(DataSet ds){
04         centerVector =new double[ds.getInputSize()];
05         for(DataSetRow row:ds.getRows()){
06              for(int i=0;i<ds.getInputSize();i++){
07                   centerVector[i]+=row.getInput()[i];
08              }
09         }
10         for(int i=0;i<ds.getInputSize();i++){
11              centerVector[i]/=ds.getRows().size();
12         }
13   }
```

上述代码中，第 6～8 行将所有样本的对应分量进行求和，第 11 行计算均值，进而求得中心向量。

对于 KohonenCos 来说，由于在学习前还必须进行样本数据的归一化以及神经元权重向量的归一化，因此重写 preLearn()函数如下：

```
1 @Override
2 public void preLearn(DataSet trainingSet) {
3    new VectorNormalizer().normalize(trainingSet);
4    randomizeWeights(new CenterVectorWeight(trainingSet));
5    randomizeWeights(new NormalizeWeight());
6 }
```

上述代码第 3 行对训练数据进行向量归一化，第 4 行根据归一化后的训练数据设置网络的初始权值，第 5 行归一化网络的权值。

2. 训练数据归一化 VectorNormalizer

归一化训练数据的 VectorNormalizer 的核心实现如下（参考公式 $\hat{X}=\frac{X}{\|X\|}$）：

```
01 public void normalize(DataSetRow row) {
02     double sum = 0;
03     for (int i = 0; i < row.getInput().length; i++) {
04         sum += row.getInput()[i] * row.getInput()[i];
05     }
06     sum=Math.sqrt(sum);
07     for (int i = 0; i < row.getInput().length; i++) {
08         row.getInput()[i]=row.getInput()[i]/sum;
09     }
10 }
```

对于每一条样本，先计算其长度（第 3～6 行），再进行拉升或者压缩成单位向量（第 7～9 行）。

3. 神经元权值向量归一化 NormalizeWeight

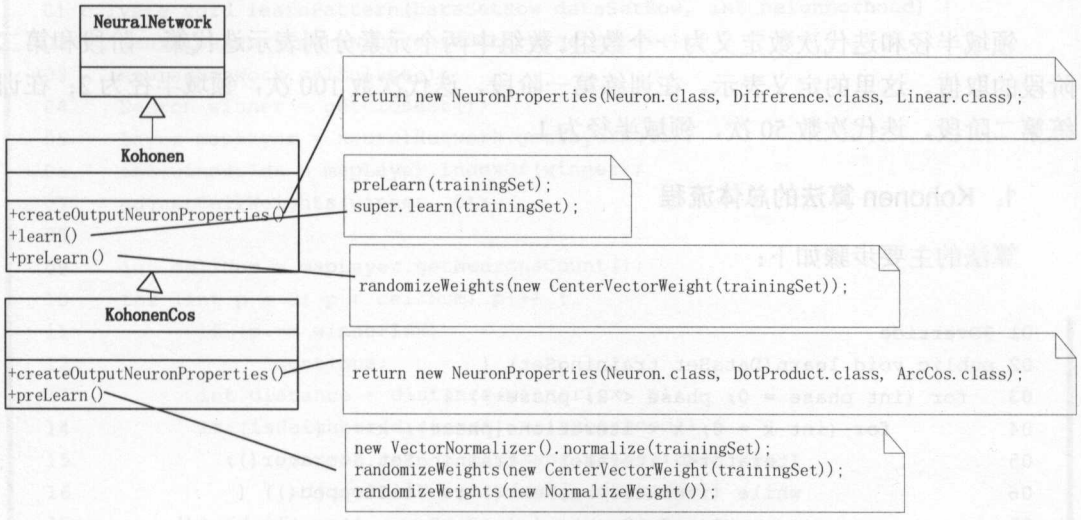
对于给定神经元，根据已有的权值，重新调整为单位向量的算法实现在 NormalizeWeight 中。其核心代码如下：

```
01 public void normalize(Neuron neuron){
02   double sum=0;
03   for (Connection connection : neuron.getInputConnections()) {
04         sum+=connection.getWeight().getValue()*connection.getWeight().getValue();
05    }
06   double mod=Math.sqrt(sum);
07   for (Connection connection : neuron.getInputConnections()) {
08         connection.getWeight().setValue(connection.getWeight().getValue()/mod)
;
09   }
10 }
```

上述代码第 3～6 行计算了权值向量的长度，第 7～9 行将权值向量转为对应的单位向量。

4. Kohonen 和 KohonenCos 的结构整理

综上所述，Kohonen 和 KohonenCos 的整体实现如图 10-7 所示。KohonenCos 的整体结构和 Kohonen 是完全一致的，差异部分则是对 Kohonen 中的函数进行了重写。



▲图 10-7 Kohonen 网络的实现

不难看出，两者的主要差异在于对于竞争层神经元传输函数和输入函数的不同，以及对训练数据和神经元权值向量归一化处理的不同。

10.3.3 Kohonen 算法的实现

虽然 Kohonen 和 KohonenCos 在结构上有少许不同，但是它们可以几乎可以共用一套 Kohonen 算法。本小节将介绍 Kohonen 算法的实现。

根据前面的描述可以知道，随着训练的进行，领域半径应该逐步减小，学习率也应该减小。为简化起见，在具体实现时，将整个训练过程分为两个阶段。第一阶段使用一个较大的领域半径和较大的学习率，第二阶段则使用较小的领域半径和较小的学习率。同时，固定第一阶段的迭代次数和第二阶段的迭代次数。当然，读者也完全可以有自己实现。比如，可以将领域半径视为迭代次数的单调递减函数。但在本书中，暂不考虑这种实现方式。

Kohonen 学习算法使用 KohonenLearning 类实现：

```
public class KohonenLearning extends LearningRule {
```

将领域半径、迭代次数、学习率申明为 KohonenLearning 类的成员：

```
double learningRate = 0.9d;
int[] iterations = { 100, 50 };
int[] nR = { 2, 1 }; // 领域半径
```

领域半径和迭代次数定义为一个数组。数组中两个元素分别表示迭代第一阶段和第二阶段的取值。这里的定义表示，在训练第一阶段，迭代次数 100 次，领域半径为 2；在训练第二阶段，迭代次数 50 次，领域半径为 1。

1. Kohonen 算法的总体流程

算法的主要步骤如下：

```
01 @Override
02 public void learn(DataSet trainingSet) {
03   for (int phase = 0; phase < 2; phase++) {
04         for (int k = 0; k < iterations[phase]; k++) {
05               Iterator<> iterator = trainingSet.iterator();
06               while (iterator.hasNext() && !isStopped()) {
07                     DataSetRow trainingSetRow = iterator.next();
08                     learnPattern(trainingSetRow, nR[phase]);
09                     if(this.neuralNetwork instanceof KohonenCos){
10                           neuralNetwork.randomizeWeights(new NormalizeWeight());
11                     }
12               } // while
13               currentIteration = k;
14               this.fireLearningEvent(new LearningEvent(this));
15               if (isStopped())
16                     return;
17         } // for k
18         learningRate = learningRate * 0.5;
19   } // for phase
20 }
```

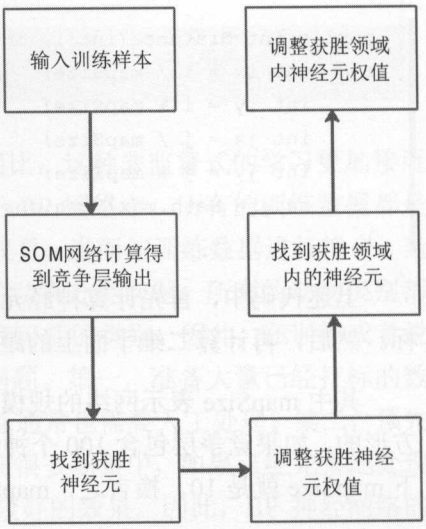
由于第二阶段训练和第二阶段训练算法步骤完全相同，只是参数不同，所以可以使同一套逻辑。上述代码第 3 行定义了训练处于第几阶段，第 4 行定义了迭代次数，第 6 行开始遍历训练数据，第 8 行的 learnPattern()函数中包含核心算法，用于具体训练一条样本。由于在 learnPattern()之后，权值向量得到了调整，不再是单位向量，所以需要在第 9～11 行进行权值向量的归一化操作。对于使用余弦相似度的 KohonenCos 网络，这个步骤是必需的。当第一阶段的训练结束后，将学习率 learningRate 减半，同时进入第二阶段的训练。

2. Kohonen 算法中对单条样本的处理

下面让我们再来看一下最为核心的 learnPattern()函数。它描述了网络如何根据一条给定的样本选取优胜节点，并调整其权值向量：

```
01 private void learnPattern(DataSetRow dataSetRow, int neighborhood) {
02     neuralNetwork.setInput(dataSetRow.getInput());
03     neuralNetwork.calculate();
04     Neuron winner = getClosest();
05     Layer mapLayer = neuralNetwork.getLayerAt(1);
06     int winnerIdx = mapLayer.indexOf(winner);
07     adjustCellWeights(winner, 0);
08
09     int cellNum = mapLayer.getNeuronsCount();
10     for (int p = 0; p < cellNum; p++) {
11           if (p == winnerIdx)
12                continue;
13           int distance = distance(winnerIdx, p);
14           if (isNeighbor(distance, neighborhood)) {
15                 Neuron cell = mapLayer.getNeuronAt(p);
16                 adjustCellWeights(cell, distance);
17           } // if
18     } // for
19 }
```

上述代码第 2 行设置网络输入。第 3 行进行计算，得到网络竞争层的输出。第 4 行根据竞争层的输出，选取与输入向量最为接近的神经元，根据网络的设计，无论是根据距离还是余弦相似度度量，这里都只需要选取输出最小的那个神经元即可。第 5～6 行得到获胜神经元的索引。第 7 行调整获胜神经元的权值。接着在第 9～18 行，调整获胜领域内的神经元权值。第 13 行，计算给定神经元和获胜神经元之间的距离。如果在获胜领域范围内，则调整这个神经元的权值向量（第 16 行）。单条样本的总体处理流程如图 10-8 所示。



▲图 10-8　根据 Kohonen 算法学习一条样本

3. 使用 getClosest()方法找到获胜神经元

根据 SOM 网络结构中对竞争层神经元输入函数

和传输函数的定义可以知道，竞争层神经元的输出或者是神经元权值向量与输入模式的距离，或者就是两者的夹角。因此，只需要简单地找到输出最小的神经元，也就是与输入模式最接近的神经元作为获胜神经元即可：

```
01 private Neuron getClosest() {
02     Neuron winner = new Neuron();
03     double minOutput = 100;
04     for (Neuron n : this.neuralNetwork.getLayerAt(1).getNeurons()) {
05         double out = n.getOutput();
06         if (out < minOutput) {
07             minOutput = out;
08             winner = n;
09         } // if
10     } // while
11     return winner;
12 }
```

上述代码第 4～10 行找到输出最小的神经元，并在第 11 行返回。

4. 计算神经元的距离 distance()与获胜领域判断 isNeighbor()

由于神经元在物理存储中，依然是使用线性方式存放，并非二维平面，因此，给出两个神经元的索引下标，还需要进行逻辑变化，才能进一步求得它们在二维平面中的实际距离。具体实现如下：

```
public int distance(int i, int j) {
    int ix = i / mapSize;
    int iy = i % mapSize;
    int jx = j / mapSize;
    int jy = j % mapSize;
    return Math.max(Math.abs(ix - jx), Math.abs(iy - jy));
}
```

上述代码中，首先计算神经元映射到二维平面后所处的位置(ix,iy)和(jx,jy)，即横纵坐标。然后，再计算二维平面上的距离。

其中 mapSize 表示网络的规模。因为竞争层的拓扑结构为一个平面，通常情况下是正方形的，如果竞争层包含 100 个神经元，那么网络结构就是 10×10 的正方形。在这种情况下 mapSize 就是 10。换言之，mapSize^2 就是竞争层神经元的个数。

得到两个神经元的距离后，就可以根据领域大小，判断一个神经元是否在另外一个神

经元的获胜领域内：

```
public boolean isNeighbor(int distance, int n) {
    return distance <= n;
}
```

其中，distance 表示神经元的实际距离，n 表示网络的获胜领域大小。

5. 获胜领域内神经元的权值调整 adjustCellWeights()

函数 adjustCellWeights()根据权值调整公式 $w_j = w_j + \alpha(\hat{X} - \hat{w}_j)$ 来调整神经元的权值：

```
1 private void adjustCellWeights(Neuron cell, int r) {
2    for (Connection conn : cell.getInputConnections()) {
3          double dWeight = (learningRate / (r + 1)) * (conn.getInput() -
           conn.getWeight().getValue());
4          conn.getWeight().inc(dWeight);
5    } // while
6 }
```

参数中 cell 表示要调整的神经元，r 表示这个神经元到获胜神经元的距离。当 r 为 0 时，表示当前调整的神经元就是获胜神经元本身。同时，前面也已经提到，随着神经元到获胜神经元之间的距离变大，学习率α应该相应地变小，两者成反比。所以，在第 3 行的实现中，使用 learningRate / $(r+1)$计算得到新的学习率来调整神经元权值向量。

10.4 SOM 网络的应用

SOM 网络属于非监督式的学习，与监督式学习相比，这种非监督式的学习更加接近机器智能。BP 神经网络是监督式学习的典型代表，在 BP 网络中，所有的训练数据都是打标数据。神经网络通过这些期望数据不断调整内部权重，进而与训练数据进行拟合。当我们把 BP 神经网络用于分类问题时，比如前面介绍的动物分类问题，所有的动物类别都是预先设定的。网络根据动物特征，将动物划分为最有可能的类别。因此，在训练或者说使用 BP 网络解决分类问题之前，不得不先解决两个问题。第一，准备大量已经打标的数据，为神经网络的训练提供必要的数据集，而数据打标通常也需要人工处理。第二，预先必须设置好需要划分的类别，而这也需要人工处理。在很多场景中，如果已经完整地处理完这两个步骤，那么 BP 网络就能依样画葫芦，取得较好的效果。因此，BP 神经网络的智力，更像是停留在“模仿”阶段，其自身不具备发现数据内在规律的能力。

而以 SOM 网络为代表的非监督式学习算法则不同，它们并不需要事先对数据打标。通过对数据的解读，它有能力挖掘数据内在联系，反而可以为人们提示数据的内部秘密。它更像是具有自身智能的学习算法。

本书将给出两个 SOM 网络的应用实例，第一是有关动物聚类，第二是将 SOM 网络应用于城市聚类。

10.4.1　使用 SOM 网络进行动物聚类

动物聚类实验是一个非常著名的 SOM 应用，它是由 Kohonen 教授本人于 1989 年给出的。在动物聚类中，训练集给出了 16 种动物以及它们的基本属性，如表 10-1 所列。

表 10-1　　动物特征

	小	中	大	2 条腿	4 条腿	毛	蹄	鬃毛	羽毛	猎	跑	飞	泳
鸽子	1	0	0	1	0	0	0	0	1	0	0	1	0
母鸡	1	0	0	1	0	0	0	0	1	0	0	0	0
鸭	1	0	0	1	0	0	0	0	1	0	0	0	1
鹅	1	0	0	1	0	0	0	0	1	0	0	1	1
猫头鹰	1	0	0	1	0	0	0	0	1	1	0	1	0
隼	1	0	0	1	0	0	0	0	1	1	0	1	0
鹰	0	1	0	1	0	0	0	0	1	1	0	1	0
狐狸	0	1	0	0	1	1	0	0	0	1	0	0	0
狗	0	1	0	0	1	1	0	0	0	0	1	0	0
狼	0	1	0	0	1	1	0	0	0	1	1	0	0
猫	1	0	0	0	1	1	0	0	0	1	0	0	0
虎	0	0	1	0	1	1	0	0	0	1	1	0	0
狮	0	0	1	0	1	1	0	0	0	1	1	0	0
马	0	0	1	0	1	1	1	1	0	0	1	0	0
斑马	0	0	1	0	1	1	1	1	0	0	1	0	0
牛	0	0	1	0	1	1	1	0	0	0	0	0	0

这里定义了 16 种动物基本属性，包括大小、腿的个数、毛发类型等。所有属性均使用 0 或者 1 表示，0 表示否定，1 表示肯定。

将这些信息作为 SOM 网络的输入，通过训练，期望得到的结果是 SOM 竞争层中的特定神经元应该会对较为接近的生物做出响应。比如，如果一个神经元对鹅进行响应，

那么它或者它周围的神经元极有可能会对鸭子。这样通过神经元的响应平面，即可归纳出这些动物的相似性。

有关动物聚类的实现，读者可以参考随书代码中的 KohonenAnimal 类。这里也将进行详细介绍。

1. 定义网络输入数据

首先作为网络的输入，需要将原始数据定义为二维数组：

```
public static double[][] data = {
            { 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0 }, // 鸽子
            { 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0 }, // 母鸡
            { 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1 }, // 鸭
            { 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 1 }, // 鹅
            { 1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 0 }, // 猫头鹰
            { 1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 0 }, // 隼
            { 0, 1, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 0 }, // 鹰
            { 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0 }, // 狐狸
            { 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0 }, // 狗
            { 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0 }, // 狼
            { 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0 }, // 猫
            { 0, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0 }, // 虎
            { 0, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0 }, // 狮
            { 0, 0, 1, 0, 1, 1, 1, 1, 0, 0, 1, 0, 0 }, // 马
            { 0, 0, 1, 0, 1, 1, 1, 1, 0, 0, 1, 0, 0 }, // 斑马
            { 0, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0 } // 牛
};
```

这组数据中，包含了每一种动物的特征信息。后续，这个 data 数组将会作为网络的输入。在一些文献资料中，还会在每一个样本前再增加 16 个分量，用于特别标识每个动物。但在本书中，不采用这种做法。

2. 定义网络竞争层的规模

```
public static final int mapSize = 4;
```

上述代码表示使用 4×4=16 个神经元作为网络输出，即输出平面大小为 4×4。

3. 定义输出字符串

```
public static String[] dataKey = {
    "鸽子", "母鸡", "鸭", "鹅", "猫头鹰",
```

```
    "隼", "鹰", "狐狸", "狗", "狼", "猫",
    "虎", "狮", "马", "斑马","牛" };
```

输出字符串 dataKey 与输入数据 data 成映射关系，即 dataKey[i]为 data[i]的描述信息。当网络给出响应后，将会在竞争平面上打印对应的 dataKey 信息，方便对数据进行观察。

4. 网络训练

网络的训练代码如下：

```
1 Kohonen som = new KohonenCos(13, mapSize * mapSize);
2 DataSet ds = new DataSet(13);
3 for (double[] row : data) {
4     ds.addRow(new DataSetRow(row));
5 }
6 som.learn(ds);
```

上述代码第 1 行定义了 SOM 网络具有 13 个输入神经元和 16 个输出神经元。这里使用 KohonenCos 网络，表示使用余弦相似度进行相似性度量。如果希望使用距离进行度量，只要简单地将 KohonenCos 替换为 Kohonen 即可。

第 2～5 行将 data 数据转为 DataSet 格式，为训练进行准备。第 6 行根据 Kohonen 算法进行学习。学习结束后，表示竞争层神经元和 data 中的数据已经建立起映射关系。

5. 打印映射关系

```
01 ResultFrame frame = new ResultFrame();
02 for (int i = 0; i < data.length; i++) {
03     som.setInput(data[i]);
04     som.calculate();
05     int winnerIndex = getWinnerIndex(som);
06     int x = getRowFromIndex(winnerIndex);
07     int y = getColFromIndex(winnerIndex);
08     System.out.println(dataKey[i] + " " + x + " " + y);
09     frame.addElementString(frame.new ElementString(dataKey[i], x, y));
10 }
11 frame.showMe();
```

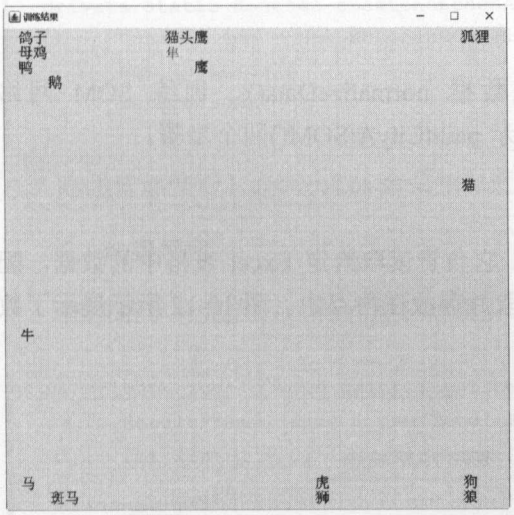
第一行申明了 ResultFrame 类。ResultFrame 将展示 SOM 竞争层神经元与输入数据的对应关系。第 3 行将某一个动物样本输入网络，第 4 行进行计算。第 5 行得到获胜神经元，也就是与输入样本对应的神经元。第 6～7 行计算获胜神经元在二维平面中的位置。第 9 行构造一个 ElementString 对象，它表示在给定 x，y 坐标下，输出 dataKey[i]，从而展示

当前输入数据在 SOM 竞争层中的对应位置。

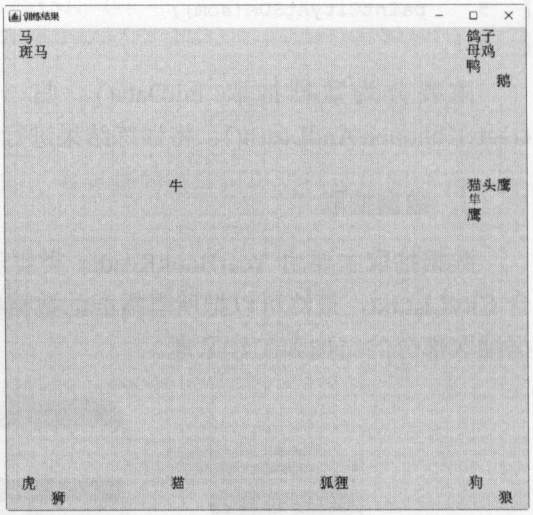
由于 ResultFrame 和 ElementString 仅仅用于界面展示，并不包含任何实质性的学习算法，所以在本书中不进行深入讨论。有兴趣的读者可以参考随书代码。

图 10-9 所示展示了由 ResultFrame 给出的一个输出。

由于初始向量的随机化，SOM 网络每次训练后给出的输出都是不一样的。图 10-10 所示展示了另一种结果。



▲图 10-9 SOM 进行动物聚类结果 1

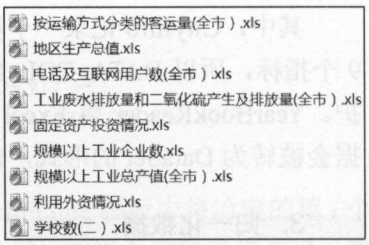


▲图 10-10 SOM 进行动物聚类结果 2

虽然两次结果不同，但不难看出，相似的动物在 SOM 网络中总是会映射到相同或者相近的神经元中，从而实现聚类。

10.4.2 使用 SOM 网络进行城市聚类

中国是一个地大物博的国家，有多达 600 个以上的城市。如何根据现有数据，合理地划分和归类这些城市，对国家和地区发展都有着重要的指导意义。本节将根据中国城市统计 2014 年年鉴的部分数据，通过 SOM 网络，对其中 43 个城市进行聚类分析。图 10-11 所示显示了在本案例中使用的 9 个数据指标。



▲图 10-11 中国城市统计 2014 年年鉴部分数据

需要提取的 43 个城市记录在随书代码的 CityList.txt

文件中。其中，包括北京、广州、杭州、沈阳等。

1. 系统主体逻辑

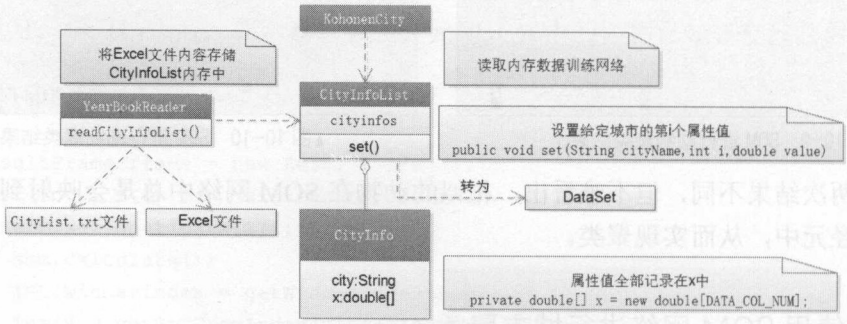
本案例中，程序由 KohonenCity 类启动。主体逻辑如下：

```
1 public static void main(String[] args) throws IOException {
2     initData();
3     normalizeData();
4     Kohonen som = createKohonenAndLearn();
5     paintCityAtSOM(som);
6 }
```

主要分为数据抽取 initData()、归一化数据 normalizeData()、训练 SOM 网络 createKohonen AndLearn()、将训练结果进行展示 paintCityAtSOM()四个步骤。

2. 数据抽取

数据抽取主要由 YearBookReader 类实现，它负责读取给定 Excel 表格中的数据。配合 CityList.txt，最终可以把所有需要的数据提取并存放在内存中。图 10-12 所示展示了数据抽取部分的结构和工作原理。



▲图 10-12　城市聚类案例的数据抽取

其中，CityInfo 记录一个城市的信息，年鉴中的数据存储在 x 数组中。在本例中使用 9 个指标，所以 DATA_COL_NUM 在运行时为 9。多个城市数据由 CityInfoList 类进行维护。YearBookReader 将 Excel 内的统计数据载入 CityInfoList。最后，CityInfoList 中的数据会被转为 DataSet 的形式，供 SOM 网络使用。

3. 归一化数据

当数据载入 DataSet 后，需要对原始数据进行预处理。所有属性值都将映射到 0～1

之间。这样做是为了防止不同属性值绝对值差距太大，从而影响训练效果。代码如下：

```
new RangeNormalizer(0,1).normalize(ds);
```

其中 ds 为含有数据的 DataSet。

4. 训练 SOM 网络

使用归一化后的数据训练神经网络：

```
private static Kohonen createKohonenAndLearn() {
      Kohonen som = new KohonenCos(CityInfo.DATA_COL_NUM, mapSize*mapSize);
      som.learn(ds);
      return som;
}
```

这里依然使用 KohonenCos 余弦相似度判断。有兴趣的读者可以尝试使用距离相似度。

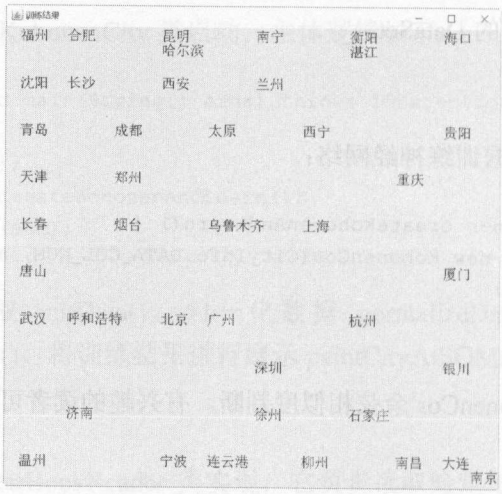
5. 结果展示

在网络学习完成后，展示其学习成果。部分代码如下：

```
01 private static void paintCityAtSOM(Kohonen som) {
02    ResultFrame frame = new ResultFrame();
03    for (int i = 0; i < data.length; i++) {
04          paintOneCity(som, frame, i);
05    }
06    frame.showMe();
07 }
08
09 private static void paintOneCity(Kohonen som, ResultFrame frame, int i) {
10    som.setInput(ds.getRowAt(i).getInput());
11    som.calculate();
12    int winnerIndex = getWinnerIndex(som);
13    int x = getRowFromIndex(winnerIndex);
14    int y = getColFromIndex(winnerIndex);
15    System.out.println(dataKey[i] + " " + x + " " + y);
16    frame.addElementString(frame.new ElementString(dataKey[i], x, y));
17 }
```

上述代码中，第 9 行定义的 paintOneCity()函数在 ResultFrame 画板中将给定的第 i 个城市标注在获胜神经元的位置上，第 10 行将第 i 个城市的属性值送入网络进行学习，第 12～14 行计算获胜神经元在二维平面中的位置。

最终，程序的输出如图 10-13 所示，图中显示了一个包含 100 个竞争神经元的 SOM 网络的输出。



▲图 10-13　包含 100 个竞争神经元的 SOM 网络

不难看出，这个聚类结果与实际也是非常相符的，如北京、广州、深圳等一线城市距离较为接近。相对地，次发达地区，如衡阳、湛江等被归为一类。此外，宁波和连云港等港口城市也在这次训练中被视为相近。

10.5 总结

本章主要介绍了竞争学习网络的原理和算法，详细阐述了自组织映射网络 SOM 的原理，包括其网络结构以及核心算法 Kohonen。SOM 网络是一种非监督神经网络，具有从数据集中挖掘隐藏信息的能力。为了展示 SOM 网络的能力，本章还给出了动物聚类以及城市聚类两个案例。从实际案例中可以看到，SOM 网络可以很好地解决聚类问题。

第11章 PCA神经网络

在数据分析时，往往需要处理极其大量的数据。随着数据规模的增加，分析的难度也会越来越大。而数据规模可以从两个角度去探讨，一是数据中包含样本的数量，二是数据的维度。比如，对于图像识别，一张200×200的图片，其数据维度达到40000。如果使用神经网络对这张图像进行学习，那就不得不构造一个包含40000个神经元输入的网络。那么网络的计算量也将是极其惊人的。为此，降低数据维度，是高维数据分析中非常重要的手段。

其次，对于高维的原始数据，我们不得不首先思考一个问题，那就是，所有这些维度都是必需的吗？答案是否定的。在大部分情况下，对于这些高维数据，往往有一些维度对于数据分析而言没有太大的意义。在分析数据时，除了带来额外的计算量外，对分析结果帮助并不大。因此，就应该避免在分析时引入这些影响不大的维度，进而降低分析难度。

主分量分析（PCA）就是为此而生的。PCA方法可以用来降低原始数据维度，提取真正有用的维度。PCA方法在形式上是一种数学方法，而PCA网络则是一种实现PCA目的的神经网络。它以神经网络的形态来解决PCA方法要解决的问题。

本章首先简要介绍PCA方法的原理和实现。其次，给出两个与PCA方法等效的PCA神经网络，并介绍PCA网络与传统的PCA方法相比有何优势。最后，将本章介绍的PCA网络应用于MNIST手写体降维。

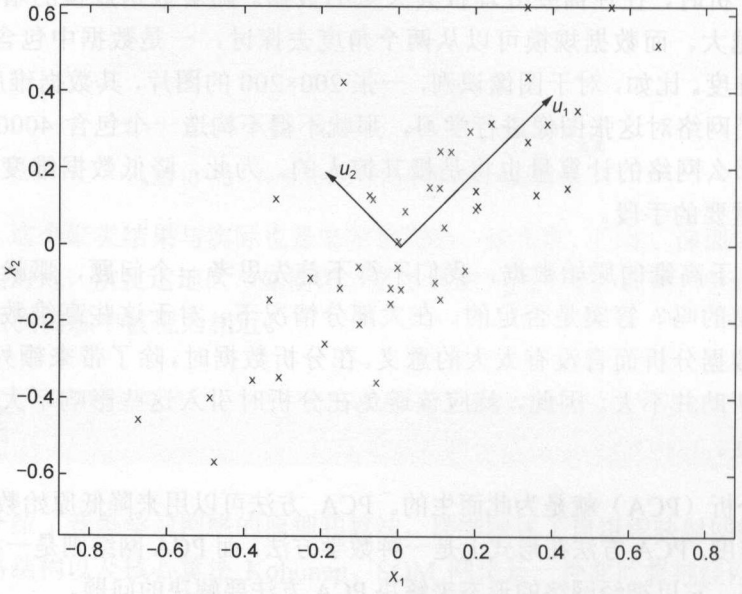
11.1 PCA方法概述

PCA全称是Principle Components Analysis，即主分量分析。它是Karhunen在1947

年提出的，Loveve 在 1963 年对其进行了归纳整理。PCA 方法包括特征选择和特征提取两个部分。其中特征选择可以看成是一个坐标变换，变换后，数据的特征会更加明显。特征提取则是在特征选择后进行的降维操作，它对变换后的向量进行截断，丢弃特征不明显的分量。PCA 方法并没有简单地对原始数据进行截断，因为我们总是希望降维后的数据特征尽可能地接近原始数据特征，而简单的数据截断无法满足这个要求。

11.1.1　PCA 方法数学背景

图 11-1 所示显示了一个 PCA 方法的几何含义。



▲图 11-1　PCA 方法的几何含义

对于二维数据(x_1,x_2)，如果要根据数据分布情况降成一维，简单地删除 x_1 或者 x_2 都无法达到良好的效果。因为数据似乎在这两个方向上的分布比重非常接近。但是，如果进行坐标变换，产生新的方向 u_1 和 u_2，不难发现，数据的分布特征在 u_1 方向上显得尤其明显，而在 u_2 上不那么明显。因此，在坐标变换后，丢弃 u_2 是一个更加合理的选择。那么 u_1 的方向应该满足什么条件呢？从直观上不难看出，所有样本点在新坐标上的投影应该尽可能离散，即在该方向上投影后的方差应该最大。换言之，方差最大的方向即为数据特征最明显的方向，也就是主分量。

1. 特征向量的选择

这里简要介绍一下特征向量选择的数学方法。令 $\boldsymbol{X}$ 为 n 维向量，并假设均值 $E(\boldsymbol{X})=0$。若 $\boldsymbol{X}$ 均值不为 0，则可以令 $\boldsymbol{X}'=\boldsymbol{X}-E(\boldsymbol{X})$，从而得到 $E(\boldsymbol{X}')=0$。

令 $\boldsymbol{U}_j$ 为 n 维单位向量，$\boldsymbol{X}$ 在 $\boldsymbol{U}_j$ 上的投影为 $y_j=\boldsymbol{U}_j^{\mathrm{T}}\boldsymbol{X}$。因为 $\boldsymbol{X}$ 的均值为 0，故 y_j 的均值也为 0，其方差为：

$$E(y_j^2)=E[(\boldsymbol{U}_j^{\mathrm{T}}\boldsymbol{X})(\boldsymbol{U}_j^{\mathrm{T}}\boldsymbol{X})]=\boldsymbol{U}_j E[(\boldsymbol{X}\boldsymbol{X}^{\mathrm{T}})]\boldsymbol{U}_j=\boldsymbol{U}_j^{\mathrm{T}}R_{XX}\boldsymbol{U}_j$$

$\boldsymbol{R}_{XX}$ 为 $\boldsymbol{X}$ 的自相关矩阵，由于 $\boldsymbol{X}$ 的均值为 0，$\boldsymbol{R}_{XX}$ 也是协方差阵。

为了使 $E(y_j^2)$ 最大，$\boldsymbol{U}_j$ 需要满足以下条件：

$$\boldsymbol{R}_{XX}\boldsymbol{U}_j=\boldsymbol{U}_j$$

即，$\boldsymbol{U}_j$ 应该是 $\boldsymbol{R}_{XX}$ 的特征向量。由于 $\boldsymbol{R}_{XX}$ 为 n 维对称阵，因此，具有 n 个非负实数特征值，即对应 n 个单位特征向量：

$$\boldsymbol{R}_{XX}\boldsymbol{U}_i=\lambda_i\boldsymbol{U}_i \quad i=1,2,3,\cdots,n$$

在计算得到 $\boldsymbol{U}_j$ 后，就可以将 $\boldsymbol{X}$ 投影到 $\boldsymbol{U}_j$ 上，并得到投影 y_j：

$$y_j=\boldsymbol{U}_j^{\mathrm{T}}\boldsymbol{X}$$

对所有的特征向量均求解，可以得到向量 $\boldsymbol{X}$ 对应投影向量 $\boldsymbol{Y}$。

2. 特征提取

在计算得到 $\boldsymbol{X}$ 的 n 个特征值 λ 后，将 λ 从大到小排列，即 $\lambda_1\geqslant\lambda_2\geqslant\lambda_3\cdots\geqslant\lambda_n$。特征值 λ 越大，表示其对应的特征向量的贡献也越大。如果希望将 n 维向量压缩到 m 维，只要简单地选取前 m 个 λ 对应的特征向量，求得对应的投影 $\boldsymbol{Y}$ 即可。前 m 个分量的方差贡献率为：

$$\frac{\sum_{i=1}^{m}\lambda_i}{\sum_{i=1}^{n}\lambda_i}=100\%$$

显然，在特征提取时，应该尽可能选择一个较小的 m，并且使得这 m 个分量的方差贡献率尽可能大。

11.1.2 PCA 计算示例

为了帮助读者更好地了解 PCA 的计算过程，这里给出一个简单的示例。在这里，将使用常规的 PCA 计算方法，对给定数据集进行 PCA 降维。同时，这里的计算结果也将作

为后续 PCA 网络输出的参考。

首先给出如下 4 个向量，通过 PCA 降维成二维向量：

x1=[2 1 5]

x2=[3 2 6]

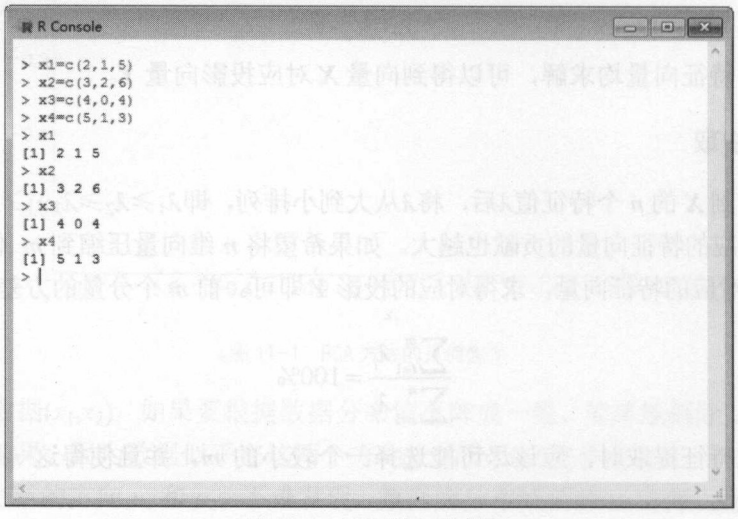
x3=[4 0 4]

x4=[5 1 3]

为方便计算，这里使用 R 语言作为计算平台。读者可以在 https://www.r-project.org/ 中下载得到 R 软件。在 R 控制台中定义上述 4 个变量：

```
>x1=c(2,1,5)
>x2=c(3,2,6)
>x3=c(4,0,4)
>x4=c(5,1,3)
```

图 11-2 所示显示了 R 语言的工作界面。



▲图 11-2　R 语言的工作界面

由于上述数据集均值非 0，所以先将其转为均值为 0 的数据集：

```
>ex=(x1+x2+x3+x4)/4
>x1=x1-ex
```

```
>x2=x2-ex
>x3=x3-ex
>x4=x4-ex
```

计算 $\boldsymbol{R_{XX}}$ 自相关矩阵，这也是协方差阵：

```
Rxx=(x1%*%t(x1)+x2%*%t(x2)+x3%*%t(x3)+x4%*%t(x4))/4
```

符号“%*%”在 R 语言中表示矩阵乘法。函数 t()表示矩阵转置。

通过 R 语言自带的 eigen()函数，求得 $\boldsymbol{R_{XX}}$ 的特征值和特征向量：

```
>eigen=eigen(Rxx)
>eigen
$values
[1] 2.3992289 0.4782026 0.1225685

$vectors
           [,1]       [,2]       [,3]
[1,]  0.6647536 -0.5842172  0.4656103
[2,] -0.2708021 -0.7693058 -0.5786492
[3,] -0.6962535 -0.2585709  0.6696060
```

可以看到，当前 $\boldsymbol{R_{XX}}$ 的 3 个特征值分别是 2.399，0.4782，0.1226。对应的 3 个特征向量也如结果所示。

最后，可以计算向量 $\boldsymbol{X}$ 在特征向量上的投影：

```
> y1=t(eigen$vectors[,1])%*%x1
> y2=t(eigen$vectors[,1])%*%x2
> y3=t(eigen$vectors[,1])%*%x3
> y4=t(eigen$vectors[,1])%*%x4
```

上述代码中，eigen$vectors[,1]表示第一个特征向量（最大特征值对应的特征向量）。

最终得到：

$$y1=-1.345257$$
$$y2=-1.647559$$
$$y3=0.9513056$$
$$y4=2.041511$$

这就是将原始数据集降维成一维后的结果。根据第二个特征向量，可以接着计算第二个分量：

```
> y1=t(eigen$vectors[,2])%*%x1
> y2=t(eigen$vectors[,2])%*%x2
> y3=t(eigen$vectors[,2])%*%x3
> y4=t(eigen$vectors[,2])%*%x4
```

得到：

$$y1=0.7470404$$
$$y2=-0.8650535$$
$$y3=0.6064826$$
$$y4=-0.4884695$$

最终，可将原始数据根据 PCA 方法压缩为二维后的结果为：

$$y1=[-1.3452570.7470404]$$
$$y2=[-1.647559-0.8650535]$$
$$y3=[0.95130560.6064826]$$
$$y4=[2.041511-0.4884695]$$

因为 3 个特征值分别是 2.399，0.4782，0.1226，因此不难得出前两个分量对于方差的贡献率为：

$$(2.399+0.4782)/(2.399+0.4782+0.1226)=95.9\%$$

11.2 PCA 神经网络学习算法

可以看出，当使用 PCA 算法进行数据数据处理时，如果维度 n 较大，在计算 $\boldsymbol{R}_{XX}$ 特征值时又会涉及矩阵相乘，而大矩阵相乘的时间复杂度为 $O(n^3)$，要精确计算高维度 n 的特征值是一件费时费力的事情。而在智能计算中，存在一种思路：当精确解很难获得时，可以退而求其次，尝试获得一个满意解。比如，在处理旅行商问题时，可以采用遗传算法、粒子群算法等。使用这些启发式算法的目的在于在有限的计算时间内，放弃寻求最优解，而尽可能地找到满意解。PCA 神经网络也是为了达到这样的目的。如果单纯使用 PCA 算法很难求解，而对数据精度又没有特别高的要求，完全可以使用满意解来替代最优解。同时，PCA 神经网络还可以根据对迭代次数的控制来控制解的精度。这里就简单介绍 Oja 和 Sanger 两种 PCA 网络算法。

11.2.1 Oja 算法

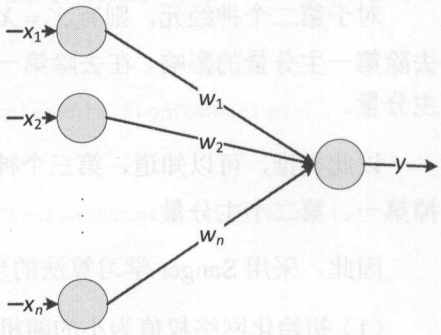
单神经元 PCA 模型结构如图 11-3 所示。

该神经元的输入为训练数据集，输出则为 $y=W^{\mathrm{T}}X=\sum_{i=1}^{n}w_i x_i$ 。

学习算法为 1982 年 E. Oja 提出的基于 Hebb 规则的 Oja 规则，该规则给出了权值调整的方式：

$$W(t+1)=W(t)+\eta[y(t)X(t)-y^2(t)W(t)]=W(t)+\eta y(t)[X(t)-y(t)W(t)]$$

其中，t 表示迭代次数，η 为学习率。该公式的特点是：当 t 趋向于无穷时，输出 y 趋向于第一主分量。对应地，权重向量 $\boldsymbol{W}$ 也趋向于特征向量 $\boldsymbol{U}$。



▲图 11-3 单神经元 PCA 网络模型

Oja 算法的具体步骤如下：

（1）初始化网络权值为小的随机数，学习率 η 通常取 0～1 之间的小数。设置网络收敛阈值 $\varepsilon>0$。这里，ε 就决定了网络的精度。

（2）对于给定样本 $X(t)$，计算网络输出 $y(t)$。

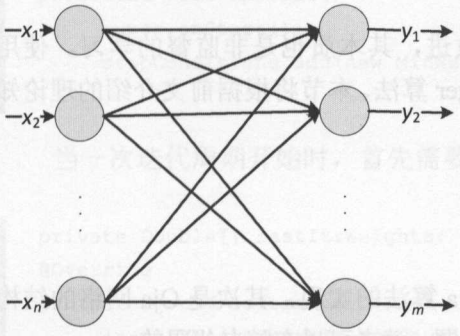
（3）根据 Oja 学习规则调整权值，得到 $\Delta W=W(t+1)-W(t)$。

（4）若 $\|\Delta W\|<\varepsilon$，则训练结束，否则继续第 2 步训练。

很明显，Oja 规则只能够将原始数据降维到一维，即计算获得第一主分量。如果我们希望使用 PCA 网络降维到任意维度，则需要使用更复杂的 Sanger 算法。

11.2.2 Sanger 算法

如果需要降维到任意维度 m（$m<n$），那么就需要图 11-4 所示的网络结构。



▲图 11-4 单层 PCA 网络结构

该模型由 T. D. Sanger 于 1989 年提出，可以将 n 维数据降维到任意 m 维。对于输出层来说有 $y_j=W_j^{\mathrm{T}}X=\sum_{i=1}^{n}w_{ij}x_i$，输出层由 m 个神经元构成。这 m 个神经元的输出最终会趋向于第一、第二，一直到第 m 个主分量。Sanger 给出该网络的权值调整公式为：

$$W(t+1)=W(t)+\eta[y(t)\hat{X}(t)-y^2(t)W(t)]=$$
$$W(t)+\eta y(t)[\hat{X}(t)-y(t)W(t)]$$

不难发现，该公式与 Oja 算法几乎一样，唯一的不同是它使用了 $\hat{X}$ 代替了 X。$\hat{X}$ 的

定义如下：

$$\hat{X} = X - \sum_{i=1}^{j-1} y_i W_i$$

该式表示第 j 个输出神经元的 $\hat{X}$ 的定义。对于第一个神经元而言，$\hat{X}=X$，也就是说第一个代表第一主分量的学习公式与 Oja 完全一致。

对于第二个神经元，则有 $\hat{X} = X - y_1 W_1$，这表示第二个神经元得到的输入向量，需要去除第一主分量的影响。在去除第一主分量后，得到的最大主分量，即为原始数据的第二主分量。

以此类推，可以知道，第三个神经元有 $\hat{X} = X - y_1 W_1 - y_2 W_2$，即第三个神经元需要去掉第一、第二个主分量。

因此，采用 Sanger 学习算法的步骤如下：

（1）初始化网络权值为小的随机数，学习率 η 通常取 0～1 之间的小数。设置网络收敛阈值 $\varepsilon>0$。这里，ε 就决定了网络的精度。

（2）对于给定样本 $X(t)$，计算网络输出 $y(t)$。

（3）根据 Sanger 学习规则调整权值，得到 $\Delta W=W(t+1)-W(t)$。

（4）若 $\|\Delta W\|<\varepsilon$，则训练结束，否则继续第 2 步训练。

通过上述训练，网络输出最终会收敛于数据的前 m 个主分量。

11.3 基于 Neuroph 实现 PCA 网络

PCA 网络的算法基于 Hebb 学习规则进行改进，其本质也是非监督的学习。使用 Neuroph 框架依然可以很顺利地实现 Oja 或者 Sanger 算法。本节将根据前文介绍的理论知识，具体实现 Oja 和 Sanger 算法。

11.3.1　Oja 算法的实现

Oja 网络的实现主要分为 3 个部分，首先是 Oja 算法的实现，其次是 Oja 网络的结构实现，最后是基于这个 Oja 网络解决简单的 PCA 问题。读者可以在随书代码的 geym.nn.pca 包中找到本章的所有源码。

1. Oja 算法的实现

Oja 算法使用 OjaLearning 类实现，OjaLearning 是基于非监督的 Hebb 规则的：

```
public class OjaLearning extends UnsupervisedHebbianLearning {
```

通过重写 UnsupervisedHebbianLearning 的部分方法，可以在学习的不同阶段自定义学习行为。在学习开始时，需要首先定义结束条件，即 ε 的值：

```
01 public class MinWeightChangeStopCondition implements StopCondition {
02   private IterativeLearninglearningRule;
03
04   public MinWeightChangeStopCondition(IterativeLearninglearningRule) {
05        this.learningRule = learningRule;
06   }
07
08   @Override
09   public booleanisReached() {
10        return delta<0.000001;
11   }
12 }
```

上述代码定义了学习结束的条件，第 10 行指定学习结束的条件为 delta<0.000001。修改这个学习条件可以调整 PCA 网络的计算量以及计算精度。

当 Oja 算法开始时，需要注册该条件使之生效：

```
@Override
protected void onStart() {
    super.onStart();
    stopConditions.add(new MinWeightChangeStopCondition(this));
}
```

当一次迭代周期开始时，首先需要记录当前的权重：

```
private Double[] lastItrWeights;
@Override
protected void beforeEpoch() {
    lastItrWeights=this.getNeuralNetwork().getWeights();
}
```

这里将当前的网络权值保存在 lastItrWeights 变量中。一次迭代学习后，便可以使用该变量求得 ΔW。

一次迭代学习过程如下：

```
1 @Override
2 public void doLearningEpoch(DataSettrainingSet) {
3    Iterator<DataSetRow> iterator = trainingSet.iterator();
4    while (iterator.hasNext() && !isStopped()) {
5          DataSetRowtrainingSetRow = iterator.next();
6          learnPattern(trainingSetRow);
7    }
8 }
```

上述代码第 6 行，根据一条训练数据，调整网络权重。函数 learnPattern()最终会调用如下代码调整网络权值：

```
01 @Override
02 protected void updateNeuronWeights(Neuron neuron) {
03   double output = neuron.getOutput();
04   for(Connection connection : neuron.getInputConnections()) {
05        double input = connection.getInput();
06        double weight = connection.getWeight().getValue();
07        double deltaWeight = (input - output*weight) * output * this.learningRate;
08        connection.getWeight().inc(deltaWeight);
09   }
10 }
```

不难看出，上述代码实现了以下公式：

$$W(t+1)=W(t)+\eta[y(t)X(t)-y2(t)W(t)]=W(t)+\eta y(t)[X(t)-y(t)W(t)]$$

其中第 7 行代码计算了 $\eta y(t)[X(t)-y(t)W(t)]$,同时在第 8 行设置了新的权重。

一次迭代学习完成后，计算$\|\Delta W\|$：

```
1 @Override
2 protected void afterEpoch() {
3    Double[] currentWeights=this.getNeuralNetwork().getWeights();
4    delta= 0;
5    for(int i=0;i<currentWeights.length;i++){
6          delta+=Math.pow((currentWeights[i]-lastItrWeights[i]),2);
```

```
7     }
8     delta=Math.sqrt(delta);
9  }
```

上述代码计算一次迭代学习后的权值变量情况，并将其保存在 delta 变量中。根据迭代学习的算法，后续会根据 stopConditions 的设置判断是否可以停止学习。

2. Oja 网络结构的定义

Oja 网络依然可以继承 NeuralNetwork 类实现：

```
public class OjaNetwork extends NeuralNetwork {
```

网络的构造过程如下：

```
01 NeuronPropertiesinputNeuronProperties = new NeuronProperties();
02 inputNeuronProperties.setProperty("neuronType", InputNeuron.class);
03
04 Layer inputLayer = LayerFactory.createLayer(inputNeuronsCount, inputNeuronProperties);
05 this.addLayer(inputLayer);
06
07 NeuronPropertiesoutputNeuronProperties = new NeuronProperties();
08 outputNeuronProperties.setProperty("transferFunction", TransferFunctionType.LINEAR);
09
10 Layer outputLayer = LayerFactory.createLayer(1, outputNeuronProperties);
11 this.addLayer(outputLayer);
12
13 ConnectionFactory.fullConnect(inputLayer, outputLayer);
14 this.randomizeWeights(new RangeRandomizer(0,1));
15 NeuralNetworkFactory.setDefaultIO(this);
16 this.setLearningRule(new OjaLearning());
```

上述代码第 1～5 行构造了网络的输入层。第 7～11 行构造了输出层，输出层只有一个神经元，并且使用线性函数作为传输函数。第 14 行将网络权值初始化为 0～1 之间的小数。第 16 行设置 Oja 学习算法。

3. 将 OjaNetwork 网络应用于实践

我们使用 OjaNetwork 用于计算本书 11.1.2 节中数据的第一主分量。首先构造数据集：

```
DataSettrainingSet = new DataSet(3, 1);
trainingSet.addRow(new DataSetRow(new double[]{-1.5,0,0.5}, new double[]{0}));
trainingSet.addRow(new DataSetRow(new double[]{-0.5,1,1.5}, new double[]{0}));
```

```
trainingSet.addRow(new DataSetRow(new double[]{0.5,-1,-0.5}, new double[]{0}));
trainingSet.addRow(new DataSetRow(new double[]{1.5,0,-1.5}, new double[]{0}));
```

注意，这里使用的是已经标准化后的均值为 0 的数据集。如果数据集均值非零，则需要先进行预处理。

接着，创建网络并设置网络的学习率为 0.01：

```
OjaNetworkoja = new OjaNetwork(3);
OjaLearninglearningRule=(OjaLearning) oja.getLearningRule();
learningRule.setLearningRate(0.01);
learningRule.addListener(this);
```

最后，使用该网络学习数据：

```
oja.learn(trainingSet);
```

学习完成后可以测试网络，即将原始数据再次输入网络，那么输出就应该是给定数据的主分量：

```
public static void testNeuralNetwork(NeuralNetworkneuralNet, DataSettestSet) {
    for(DataSetRowtestSetRow : testSet.getRows()) {
        neuralNet.setInput(testSetRow.getInput());
        neuralNet.calculate();
        double[] networkOutput = neuralNet.getOutput();

        System.out.print("Input: " + Arrays.toString( testSetRow.getInput() ) );
        System.out.println("Output: " + Arrays.toString( networkOutput) );
    }
}
```

执行上述代码，一次可能的输出如下：

```
Input: [-1.5, 0.0, 0.5]Output: [1.3479060756306194]
Input: [-0.5, 1.0, 1.5]Output: [1.6480335151179977]
Input: [0.5, -1.0, -0.5]Output: [-0.9489540544783677]
Input: [1.5, 0.0, -1.5]Output: [-2.0469855362702494]
```

细心的读者可能会发现，这个输出与理论上通过 PCA 方法得到的输出有差异，即 PCA 网络的输出均为理论 PCA 结果的相反数。如果多运行几次代码，读者也能发现 PCA 神经网络还是有很高的概率可以与理论 PCA 的计算结果完全一致。这是因为 PCA 网络的初始

权值为随机数，网络的学习会被随机化，因此可能会出现两个不同的方向发展。但值得一提的是，无论是结果与理论 PCA 方法一致还是得到了相反数，都不会影响对数据的后续处理。因为在这个问题上，我们只是希望找出方差最大的方向，显然对于 x 方向或者 x 负方向而言，其方差分布式完全一致的。

11.3.2 Sanger 算法的实现

从算法实现上说，Sanger 算法实际上是 Oja 算法的扩展。因此，就具体实现上来说两者具有相同的框架结构。所以在此只介绍两者不同之处，完整代码请读者参考随书代码。

Sanger 学习算法使用 SangerLearning 类实现。与 Oja 相比，其最大不同在于如何更新神经元权重：

```
01 @Override
02 protected void updateNeuronWeights(Neuron neuron) {
03   double output = neuron.getOutput();
04   int i = index(neuron);
05   for (int k = 0; k <neuron.getInputConnections().length; k++) {
06         Connection connection = neuron.getInputConnections()[k];
07         double input = connection.getInput();
08         double yw = 0;
09         //前 n 个神经元 Y*W 之和
10         for (int n = 0; n < i; n++) {
11               yw += index(n).getOutput() * index(n).getInputConnections()[k].
                   getWeight().getValue();
12         }
13         input -= yw;
14         double weight = connection.getWeight().getValue();
15         double deltaWeight = (input - output * weight) * output * this.learningRate;
16         connection.getWeight().inc(deltaWeight);
17   }
18 }
```

上述代码第 4 行，获得给定神经元在输出层的位置。第 9～12 行，计算当前神经元之前的 n 个神经元的 YW 之和。第 11 行的 index()函数返回给定位置上的输出神经元。第 13 行计算得到 input，也就是 Sanger 算法中的 $\hat{X}$。第 14～15 行，根据计算得到的 $\hat{X}$ 以及 Sanger 规则计算ΔW。

Sanger 网络结构的创建和 Oja 网络十分类似，这里不再重复。尝试使用 Sanger 网络

计算本书 11.1.2 节中训练数据的第一、第二主分量，得到：

```
Input: [-1.5, 0.0, 0.5]Output: [-1.3479145866485949, -0.7365598696729038]
Input: [-0.5, 1.0, 1.5]Output: [-1.6480233829864401, 0.8671997075083833]
Input: [0.5, -1.0, -0.5]Output: [0.9489467473794008, -0.6077470927322607]
Input: [1.5, 0.0, -1.5]Output: [2.0469912222556346, 0.4771072548967812]
```

可以看到 Sanger 网络的输出符合预期。

11.4 使用 PCA 网络预处理 MNIST 手写体数据集

在本书第 7 章中，已经介绍了使用 BP 神经网络识别 MNIST 手写体的问题。由于每张图片大小为 28×28，因此直接处理这些图片时，需要 784 个神经元。如果借助 PCA 网络，则完全可以在 BP 神经网络处理原始数据之前，先将数据降维，然后使用 BP 网络处理降维后的数据。从更广泛的意义上说，在分析这些高维度数据之前，先使用 PCA 网络降维，再对降维后的数据进行分析，有助于降低后续的数据分析成本。

在本节中，将尝试使用 Sanger 网络将 784 维度的输入压缩到 20 维，然后再使用 BP 网络处理压缩后的数据。

首先，定义压缩的目标维度为 20：

```
public static int dim = 20;
```

读取训练数据，并预处理数据，将均值设置为 0：

```
1 DataSettrainingSet = MnistReader.trainingData(
2     MnistTraining.MnistDir + "\\train-images.idx3-ubyte",
3    MnistTraining.MnistDir + "\\train-labels.idx1-ubyte", 60000);
4
5 makeAveZero(trainingSet);
```

上述代码 1～3 行读取 MNIST 训练数据，第 5 行预处理训练数据。函数 makeAveZero() 的实现如下：

```
01 public void makeAveZero(DataSettrainingSet) {
02   intinputCount = trainingSet.getRowAt(0).getInput().length;
03   introwCount = trainingSet.getRows().size();
04   double[] ave = new double[inputCount];
```

```
05    for (int i = 0; i <rowCount; i++) {
06          for (int j = 0; j <inputCount; j++) {
07                ave[j] += trainingSet.getRowAt(i).getInput()[j];
08          }
09    }
10
11    for (int i = 0; i <ave.length; i++) {
12          ave[i] /= rowCount;
13    }
14
15    for (int i = 0; i <rowCount; i++) {
16          for (int j = 0; j <inputCount; j++) {
17                trainingSet.getRowAt(i).getInput()[j] -= ave[j];
18          }
19    }
20 }
```

上述代码第 5～9 行，对数据根据列进行求和，第 11～13 行计算每一列的均值，第 15～19 行将均值回写到数据集中。

构造 Sanger 网络，并进行训练：

```
1 SangerNetwork sanger = new SangerNetwork(trainingSet.getRowAt(0).getInput().length, dim);
2
3 SangerLearninglearningRule = (SangerLearning) sanger.getLearningRule();
4 learningRule.setLearningRate(0.001);
5 learningRule.setMaxIterations(6);
```

上述代码第 1 行，创建 Sanger 网络，具有 784 个输入和 20 个输出。第 4 行设置网络的学习率为 0.001。第 5 行设置网络的最大迭代次数为 6 次。

训练 Sanger 网络：

```
sanger.learn(trainingSet);
```

训练完成后，该 Sanger 网络就具备了将给定 784 维的样本数据压缩至 20 维的能力。因此，压缩原始数据至 20 维：

```
DataSetpcaDateSet = pca(sanger, trainingSet);
```

上述 pca()方法完成了数据压缩。输入为 784 位 trainingSet 原始数据，输出为 20 维 pcaDateSet 数据。其中，pca()方法的实现如下：

```
1 public static DataSetpca(NeuralNetwork sanger, DataSettestSet) {
2    DataSettrainingSet = new DataSet(dim, 10);
3    for (DataSetRowtestSetRow :testSet.getRows()) {
4          sanger.setInput(testSetRow.getInput());
5          sanger.calculate();
6          trainingSet.addRow(new DataSetRow(sanger.getOutput().clone(), testSetRow.
           getDesiredOutput().clone()));
7    }
8    return trainingSet;
9 }
```

在该方法中，对每一条样本数据进行处理，第 5～6 行得到 Sanger 网络输出作为新的样本数据。

将由 pca()方法计算生成的新的 20 维样本作为训练数据，用以训练 BP 神经网络：

```
01 MlPerceptronmyMlPerceptron = new MlPerceptron(TransferFunctionType.SIGMOID, dim, 40, 10);
02 // 设置可接受的误差
03 BackPropagation learningRule1 = myMlPerceptron.getLearningRule();
04 learningRule1.setLearningRate(0.01);
05 learningRule1.setMaxError(0.05d);
06 learningRule1.setMaxIterations(100);
07 learningRule1.addListener(this);
08
09 System.out.println("Training neural network...");
10 myMlPerceptron.learn(pcaDateSet);
```

第 1 行，构造神经网络，拥有 20 个输入、40 个隐层节点和 10 个输出神经元。第 4～7 行设置 BP 神经网络的参数。第 10 行进行学习。

训练结束后，使用 MNIST 提供的测试数据集测试网络的性能。

首先，读取测试数据集，并将其预处理为均值为 0 的数据：

```
DataSettestDataSet = MnistReader.trainingData(
    MnistTraining.MnistDir + "\\t10k-images.idx3-ubyte",
    MnistTraining.MnistDir + "\\t10k-labels.idx1-ubyte", 10000);

makeAveZero(testDataSet);
```

接着，使用 Sanger 网络压缩测试数据集：

```
DataSetpcaDateSet = pca(sanger, testDataSet);
```

将压缩后的测试数据用于测试 BP 网络：

```
01 intrightCount = 0;
02 int i = 0;
03 for (; i <pcaDateSet.size(); i++) {
04    neuralNet.setInput(pcaDateSet.getRowAt(i).getInput());
05    neuralNet.calculate();
06    double[] networkOutput = neuralNet.getOutput();
07    // 将活跃度最高的输出视为 1，其余视为 0
08    networkOutput = Utils.competition(networkOutput);
09    if (Arrays.equals(networkOutput, testDataSet.getRowAt(i).getDesiredOutput()))
{
10          rightCount++;
11    }
12    if (i % 1000 == 0)
13          System.out.println("正确率:" + rightCount * 1.0 / (i + 1));
14 }
15 System.out.println("正确率:" + rightCount * 1.0 / (i + 1));
```

上述代码中，第一行 rightCount 变量用以保存判断正确的样本数量。第 4 行取得一条样本，第 5 行将该样本送入 BP 网络，第 6 行得到网络输出，第 8 行取得最为活跃的神经元作为网络输出，其他神经元则进行抑制。第 9 行判断 BP 网络输出与期望值是否相符，第 10 行进行正确率的统计。最终，在第 15 行输出整体正确率。

执行上述代码，最终得到的正确率为：

```
正确率:0.9459054094590541
```

可以看到，在进行 PCA 压缩后，BP 网络的正确率并未出现明显下降，也就说现有的 20 维数据几乎包含了原始 784 维数据的所有信息，压缩并不影响数据分析的效果。

因此，在对数据挖掘的研究中，完全可以通过 PCA 方法先压缩原始数据集，再以压缩后的数据为研究对象，应用于其他各种形式的算法和模型，进而找出最合理的算法。这样，不仅不会影响算法模型的效果，而且能够大大降低计算量。

11.5 总结

本章主要介绍了 PCA 神经网络。首先，对传统的 PCA 方法进行了回顾，接着提出了

PCA 网络的概念和算法。与传统的 PCA 方法相比，PCA 神经网络在取得类似效果的同时，还可以根据应用的需要调整算法的精度，从而降低计算工作量。而这是传统 PCA 方法无法做到的。此外，本章还详细阐述了 Oja 规则和 Sanger 规则，并使用 Neruoph 框架实现了这两个网络。最后，作为实际使用案例，我们使用 Sanger 网络对 MINIST 数据集进行降维，并取得了良好的实验效果。

欢迎来到异步社区！

异步社区的来历

异步社区（www.epubit.com.cn）是人民邮电出版社旗下 IT 专业图书旗舰社区，于 2015 年 8 月上线运营。

异步社区依托于人民邮电出版社 20 余年的 IT 专业优质出版资源和编辑策划团队，打造传统出版与电子出版和自出版结合、纸质书与电子书结合、传统印刷与 POD 按需印刷结合的出版平台，提供最新技术资讯，为作者和读者打造交流互动的平台。

社区里都有什么？

购买图书

我们出版的图书涵盖主流 IT 技术，在编程语言、Web 技术、数据科学等领域有众多经典畅销图书。社区现已上线图书 1000 余种，电子书 400 多种，部分新书实现纸书、电子书同步出版。我们还会定期发布新书书讯。

下载资源

社区内提供随书附赠的资源，如书中的案例或程序源代码。

另外，社区还提供了大量的免费电子书，只要注册成为社区用户就可以免费下载。

与作译者互动

很多图书的作译者已经入驻社区，您可以关注他们，咨询技术问题；可以阅读不断更新的技术文章，听作译者和编辑畅聊好书背后有趣的故事；还可以参与社区的作者访谈栏目，向您关注的作者提出采访题目。

灵活优惠的购书

您可以方便地下单购买纸质图书或电子图书，纸质图书直接从人民邮电出版社书库发货，电子书提供多种阅读格式。

对于重磅新书，社区提供预售和新书首发服务，用户可以第一时间买到心仪的新书。

用户账户中的积分可以用于购书优惠。100 积分 =1 元，购买图书时，在 0 使用积分 里填入可使用的积分数值，即可扣减相应金额。

特 别 优 惠

购买本书的读者专享异步社区购书优惠券。

使用方法：注册成为社区用户，在下单购书时输入 S4XC5 使用优惠码，然后点击“使用优惠码”，即可在原折扣基础上享受全单9折优惠。（订单满39元即可使用，本优惠券只可使用一次）

纸电图书组合购买

社区独家提供纸质图书和电子书组合购买方式，价格优惠，一次购买，多种阅读选择。

社区里还可以做什么？

提交勘误

您可以在图书页面下方提交勘误，每条勘误被确认后可以获得100积分。热心勘误的读者还有机会参与书稿的审校和翻译工作。

写作

社区提供基于Markdown的写作环境，喜欢写作的您可以在此一试身手，在社区里分享您的技术心得和读书体会，更可以体验自出版的乐趣，轻松实现出版的梦想。

如果成为社区认证作译者，还可以享受异步社区提供的作者专享特色服务。

会议活动早知道

您可以掌握IT圈的技术会议资讯，更有机会免费获赠大会门票。

加入异步

扫描任意二维码都能找到我们：

异步社区	微信服务号	微信订阅号	官方微博	QQ群：436746675

社区网址：www.epubit.com.cn

投稿&咨询：contact@epubit.com.cn

作者简介 | **葛一鸣**

硕士，国家认证系统分析师，Oracle OCP。长期从事 Java 软件开发工作，对 Java 技术、人工智能、神经网络、数据挖掘等技术有浓厚兴趣。现著有《自己动手写神经网络》（百度阅读电子书）、《Java 程序性能优化》《实战 Java 虚拟机》《实战 Java 高并发程序设计》。

异步社区
人民邮电出版社
www.epubit.com.cn

异步社区 www.epubit.com.cn
新浪微博 @人邮异步社区
投稿/反馈邮箱 contact@epubit.com.cn

封面设计：广领设计
分类建议：计算机／计算机科学／人工智能
人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-46201-5
ISBN 978-7-115-46201-5
定价：55.00 元